UNIVERSIDADE da MADEIRA

Universidade da Madeira
Centro de Ciencias Exactas e da Engenharia

PhD THESIS

presented in fulfilment of the requirement for
the degree of Doctor of Philosophy

Major: **Software Engineering**

Code Generation for Event-B

presented by

VÍCTOR ALFONSO RIVERA ZÚÑIGA

supervised by

NÉSTOR CATAÑO COLLAZOS

June 2014

JURY

Timothy Wahls	Dickinson College, USA	Reviewer
Camilo Rueda	Javeriana University, Colombia	Reviewer
Pedro Campos	University of Madeira, Portugal	Reviewer
José Carmo	University of Madeira, Portugal	Rector
Néstor Cataño	Madeira-ITI, Portugal	Supervisor

UNIVERSIDADE da MADEIRA

Code Generation for Event-B

To my parents and Laura N. Mazaira.

Acknowledgments

The fulfilment of this thesis would not have been possible without the contribution of Néstor Cataño, my thesis supervisor. I want to thank him for his guidance during all this process, his encouragement, and dedication. Specially thanks for his support and patience. I also want to thank Camilo Rueda and Tim Wahls for their ideas, and discussions. Thanks to Madeira Interactive Technologies Institute (M-ITI - Portugal) and *Departamento Administrativo de Ciencia, Tecnología e Innovación* (Colciencias - Colombia) for funding my research over these years. Many thanks, from the bottom of my heart, to my mother Esperanza Zúñiga (R.I.P.) and my father Luis Rivera for their love and their enthusiasm that have carried me out this far.

I apologize to all those not mentioned that have helped me in one way or another. Even though I have unconsciously omitted their names, I am very grateful for the help they provided.

This work was supported by the Portuguese Foundation for Science and Technology (FCT) grant PTDC/EIA-CCO/105034/2008 (FAVAS: A FormAl Verification for real-time Systems), and Colciencias, The Colombian Agency for Science and Technology Development.

Contents

List of Figures

Abstract

Refinement Calculus and Design-by-Contract are two formal methods widely used in the development of systems. Both methods have (dis-)advantages: in Refinement Calculus a model starts with an abstraction of the system and more details are added through refinements. Each refinement is probably consistent with the previous one. Hence, the reasoning about abstract models is possible. However, a high level of expertise is necessary in mathematics to have a good command of underlying languages, techniques and tools, making this methodology less popular; on the other hand, Design-by-Contract works on the program rather than the program model so developers in software industry are more likely to have expertise on it. However, the benefits of reasoning over more abstract models are lost.

A question arises, is it possible to combine both methods so users can use them together in the development of systems taking the best of both? This thesis solves this question by translating Refinement Calculus with Event-B to Design-by-Contract with Java and JML so users can take full advantage of both formal methods without losing their benefits. This thesis presents a set of syntactic rules that translates Event-B to JML-annotated Java code. It also presents the implementation of the syntactic rules as the EventB2Java tool. We used EventB2Java to translate several Event-B models. The tool generated JML-annotated Java code for all the considered Event-B models that serve as final implementation. We also used EventB2Java for the development of two software applications. Additionally, we compared EventB2Java against two other tools that also generate Java code from Event-B. EventB2Java enables users to start the development of software in Event-B where users can model the system and prove it consistently, to then transition to JML-annotated Java code using our tool, where users can continue the development.

Key Words— Refinement Calculus, Event-B, Design-by-Contract, Java, JML, EventB2Java.

Resumo Português

Refinement Calculus e *Design-by-Contract* são dois métodos formais utilizados no desenvolvimento de sistemas. Ambos têm (des-)vantagens: em *Refinement Calculus* um modelo começa com uma abstração do sistema e mais detalhes são adicionados através de refinamentos. Cada refinamento é provavelmente consistente com o anterior. Assim, o raciocínio sobre modelos abstratos é possível. No entanto, um alto nível de conhecimento é necessário em matemática para ter um bom domínio da sintaxe da linguagem, técnicas e ferramentas tornando esta metodologia menos popular; *Design-by-Contract* trabalha no programa, e não no modelo do programa de modo que os desenvolvedores na indústria de software são mais predispostos a ter conhecimento sobre ele. No entanto, os benefícios do raciocínio em relação aos modelos mais abstratos são perdidos.

Surge uma questão, é possível combinar ambos métodos para que os utilizadores possam utilizá-los conjuntamente no desenvolvimento de sistemas tomando o melhor dos dois? Esta tese resolve essa questão, traduzindo *Refinement Calculus* com Event-B para o *Design-by-Contract* com Java e JML assim é possível tirar o máximo proveito de ambos métodos formais. Apresenta-se aqui um conjunto de regras sintáticas que traduz Event-B para o código Java com anotações JML. Apresenta também a implementação das regras sintáticas como a ferramenta EventB2Java. Usamos EventB2Java para traduzir vários modelos em Event-B. A ferramenta gerou código Java com anotações JML para todos os modelos considerados que servem de implementação final. Também usamos EventB2Java para o desenvolvimento de duas aplicações de software. Além disso, comparamos EventB2Java com duas outras ferramentas que também geram código Java do Event-B. EventB2Java permite que os utilizadores iniciem o desenvolvimento de software em Event-B onde podem modelar o sistema, prová-lo de forma consistente e fazer a transição para código Java com anotações JML utilizando a nossa ferramenta, onde os utilizadores podem continuar o desenvolvimento.

Palavras-chave— Refinement Calculus, Event-B, Design-by-Contract, Java, JML, EventB2Java.

Chapter 1

Introduction

Information systems have become essential to people. As an example of this, people use web systems to search for things to buy, and use bank transaction systems to make payments, and even trust their lives to critical software systems, such as control software used by airplanes. Often, people are unaware of the consequences that malfunctioning software can do to their lives. Hence, they must be built in a correct fashion.

Concepts such as robustness and reliability are important in software today: people expect software systems to work well. Several approaches to software reliability and robustness exist [53, 89], which might be supplemented with Software Testing [13, 14]. However, testing techniques alone are not adequate to ensure the correctness of a critical software (or any other software). As Edsger Dijkstra said "Testing shows the presence, not the absence of bugs". Tests can only show the situations where a system will fail, but cannot say anything about the behaviour of the system outside the testing scenarios. While it is true that validating the code against certain properties, as in testing, makes software testing popular and important, it is also true that testing does not validate the system as a whole: system behaviours beyond the ones considered by the tests can produce casualties.

A way to ensure the correctness of critical software is using formal methods [29] which are mathematically based on rigorous techniques for the specification of systems (well-formed statements in a mathematical logic) from requirements, the verification (rigorous deductions in that logic), and the implementation of software (and hardware) systems. Formal methods enable users to express properties over the system that must be proven true for all possible inputs. Formal methods are concerned with the system as a whole, proving that each component of the system interacts with each other in a correct way. It seems right then to think that formal methods are the key for the construction of correct software. Testing can be seen as a complement of formal methods: one can model a system using formal methods ensuring the correctness of the system w.r.t. some requirements and then testing to be sure that what one proved mathematically was indeed what one wanted.

Refinement Calculus techniques [52, 80, 79, 9] are techniques to implement software systems based on formal methods. In Refinement Calculus, users write an abstract model of a system, and then transform the model into an implementation via a series of refinement steps, where each refinement adds more detail to the system. The behaviour of each refinement is provably consistent with the behaviour of the previous step. The final refinement is the actual implementation of the system modelled. This technique is known as correctness-by-construction, as it allows to reasoning about the model ensuring that all functionality is correct. B [2] and Event-B [4] are examples of formal techniques based on Refinement Calculus: B is a method for specifying, designing and coding software systems introduced by J.-R. Abrial. Atelier B [8] is an IDE that enables user to work with B method; Event-B models are complete developments of discrete transition systems. Event-B was also introduced by J.-R. Abrial and it is derived from the B method. Rodin [28] is an Eclipse IDE that provides support for Event-B. The correctness of models in B and Event-B is achieved by discharging proof obligations. Proof obligations are correctness conditions on the model of the system which need to be proven true. Several provers exist that help the process of discharging proof obligations. For instance, Atelier B (for working with B) comes with its own automatic prover. Rodin (for working with Event-B) comes also with its own automatic prover. Other provers could be used in the process of discharging proofs obligations. For example, Dafny [65] is an imperative object-based language with built-in specification constructs that comes with an automatic prover, Z3 [49], one could express proof obligations in the input language of Dafny and then use its automatic provers to discharge the proof obligations.

Limitations of Refinement Calculus techniques come from the level of expertise in mathematics required to use the underlying languages and tools. As J. Bowen and V. Stavridou describe in [24], one of the principal issues of the wide adoption of formal methods is that they require mathematical expertise. For instance, when applying refinement techniques, making the behaviour consistent of a refinement with the behaviour of the previous step is heavy burden, particularly when the refinement is close to the implementation. Developers of software are not correctly educated (as states by J.-R. Abrial in [4]), since they have little or no mathematical background, making Refinement Calculus complex to use.

Another technique to implement software systems based on formal methods is Design-by-Contract (DbC) [78]. The general idea about Design-by-Contract is that a software contract exists between a method and a client. The client must fulfil the pre-condition of the method that is called and this must ensure its post-condition. Design-by-Contract techniques work on the program rather than on the program model, so developers in software industry are more likely to have expertise in this technique. Java Modeling Language (JML) [70, 25, 71] is based on Design-by-Contract in which code is verified against a formal specification. Likewise Refinement Calculus, DbC techniques have some limitations, Design-by-Contract does not have the mathematical based rigorous as in Refinement Calculus, so reasoning over more abstract models is lost.

The main goal of this thesis is to bridge Refinement Calculus with Event-B to Design-by-Contract with Java and JML. Thus, software developers can benefit from both formal methods in the software development of applications. This thesis presents a code generator for Event-B that generates JML-annotated Java code. Thus, the software development of an application starts with a formal model in Event-B on which users can define properties on that can be proven correct by discharging proof obligations in Event-B. Then, the user translates the model to a JML-annotated Java code using our code generator. The user decides the level of model abstraction in Event-B. Once the code is generated, the user can continue the system development of the application in Java.

The work presented in this thesis allows users of different expertise to work together in the development of systems. For instance, a user expert in the notation underlying Event-B and expert in mathematics can work at early stages of the system development, then transition to JML-annotated Java code of the Event-B model using our code generator, where an expert software developer can continue with the final implementation of the system.

The code generator from Event-B to JML-annotated Java code is formally defined by means of translation rules. This thesis presents the translation rules and its implementation as the EventB2Java tool. EventB2Java automates the process of code generation from Event-B models. The code generator, additionally to generating Java code of Event-B models, also generates JML specifications. Thus, users can customise the Java code and verify it against the JML specifications to make sure the customisation does not invalidate the initial Event-B model.

We have validated EventB2Java by using the tool in the process of generating JML-annotated Java code from an ample set of Event-B models. This thesis presents these Event-B models and the code generated by EventB2Java, and also presents a benchmark in which EventB2Java is compared against other tools for generating Java code from Event-B models. We used EventB2Java in the development of two case studies: the first case study is on the development of an Android [61] application that follows the MVC (Model-View-Controller) design pattern [60]; the second case study is on testing an Event-B model by translating it to Java and performing Java Unit (JUnit) testing of the generated Java code.

This thesis also introduces a tool that generates Dafny code from Event-B proof obligations (PO) to assist users in the process of proving the system correct. The translation of Event-B POs to Dafny is defined by means of translation rules. Rules were implemented as the EventB2Dafny tool. EventB2Dafny helps users on discharging Event-B POs by translating them to the input language of Dafny where the user can use Z3, the Dafny automatic prover to prove the PO.

Thesis Summary: The development of the work presented in thesis started by proposing a translation from B machines to JML specifications (described in Chapter 3). We saw B method a good starting candidate for the development of systems since systems are first modeled in an abstract way. Next, the model is proven to satisfy certain safety and security properties, and then transformed to code via a series of property preserving refinement steps. We proposed the B2Jml tool that generates JML specifications from B, where users can manually write Java code. Then we realised that Event-B method (an evolution of the B method) is a better starting candidate for the development of systems, so we proposed a translation from Event-B machines to JML specifications. Chapter 4 discusses how one can see Event-B method better than B, the chapter also discusses the translation and the implementation of the EventB2Jml tool. In modelling a system in Event-B, one needs to prove the soundness of the model. A series of proof obligations are generated and needed to be discharged to gain confidence of the model. Discharging proof obligation can be a difficult task, so we proposed a translation from Event-B proof obligations to the input language of Dafny, thus users can use Dafny as a prover. Our intention is to provide tools that help users in the process of proving an Event-B model sound. Chapter 6 describes this translation and presents the EventB2Dafny tool that automates the process of translation. Once we developed this tool and EventB2Jml, we used them to generate JML specifications from the Event-B model MIO. Then we manually generated Java code for those specifications. We then realised that is more useful to have a tool that automatically generates Java code and we saw the importance of embedded JML specifications into the code since users might want to customise the generated code without invalidating the initial model. We proposed and implemented (discussed it in Chapter 5) a translation from Event-B models to JML-annotated Java code. We implemented it as the EventB2Java tool. Having the JML specifications embedded into the Java code also gives a insight of documentation of the code that can be read easily.

1.1 Thesis overview

Chapter §1. Introduction This chapter describes the problem addressed by this thesis and describes the importance of using more-formal methods to build systems

Chapter §2. Background This chapter provides the background knowledge required to understand the work done in this thesis.

Chapter §3. Translating B Machines to JML Specifications. This thesis work started with the idea of generating JML specifications from B. This chapter is about that. The translation is defined using syntactic rules and it is implemented as the B2Jml tool, and integrated to ABTools [18] (an open source environment for developing B). B2Jml enables users to use B's strong support for model verification during early stages of software development to generate a fully verified model of an application, and then transition to JML specifications to simplify the task of generating a Java implementation and to take advantage of JML (semi-) automatic tools such as runtime or static assertion checkers.

Contributions. The main contributions of this chapter are *i*) the definition of a set of sound rules to translate B to JML, *ii*) the implementation of the rules as the B2Jml tool. Work done in this chapter have been published in [41, 36]. I participated in the design and testing of the syntactic rules and in the design of B2Jml and its integration to the ABTools suite.

Chapter §4. Translating Event-B Machines to JML Specifications. This chapter presents a translation from Event-B to JML. This work goes in the same direction as the work presented in Chapter 3 as they both generate JML specifications from a formal model. I decided to change the initial formal method from B to Event-B since Event-B presents a simpler notation than B, which is easier to learn, use, and translate. In addition to this, the B method is devoted to the development of correctness-by-construction software, while the purpose of Event-B is used to model full systems (including hardware, software and environment of operation). Chapter 4 presents a set of syntactic rules to generate JML specifications from Event-B. It presents the implementation of the rules as the EventB2Jml tool. EventB2Jml is implemented as a Rodin [28] plug-in. This chapter also shows the application of the tool to a model in Event-B.

Contributions. The main contributions of this chapter are the definition of a translation from Event-B models to JML specifications, and the implementation of the translation as the EventB2Jml tool. This translation allows experts in Event-B to work together with software developers usually expert in main stream programming languages like Java. For instance, an expert in the notation underlying Event-B starts the development of a system, then uses EventB2Jml to transition to JML where a software developer writes Java code from the JML specifications. I participated in the definition of the syntactic rules of the translation and I fully implemented the EventB2Jml tool.

Chapter §6. Translating Event-B Machines Proof Obligations to Dafny. This chapter presents a translation of Event-B proof obligations to the input language of Dafny by means of syntactic rules, Dafny [65] is an imperative object-based language with built-in specification constructs. The rules were implemented as the EventB2Dafny Rodin plug-in. To prove an Event-B model consistent it is necessary to discharge a serie of proof obligations. Typically, proof obligations are automatically discharged by Rodin provers. However, there are some proofs that need users' assistance to be discharged. EventB2-Dafny assists users on the process of discharging proof obligations by translating them to Dafny. EventB2Dafny generates Dafny code that is correct if and only if the Event-B refinement-based proof obligations hold.

Contributions. The main contributions of this chapter are the definition of a translation from Event-B proof obligations to Dafny and the implementation of the translation as the EventB2Dafny tool. Work done in this chapter has been published in [35]. I participated in the definition of the translation from Event-B proof obligations to Dafny programming language and in the implementation of EventB2Dafny.

Chapter §5. Translation of Event-B Machines to JML-annotated Java Code. We decided to extend the work described in Chapter 4 to not just generate JML specifications, but also to generate Java code from Event-B models. This chapter presents the core work of this Ph.D. thesis. It describes the translation of Event-B models to JML-annotated Java code. The translation is achieved through syntactic rules and it is implemented as the EventB2Java tool.

Contributions. The main contributions of this chapter are *i*) the definition of a set of sound rules to translate Event-B models to JML-annotated Java code, *ii*) the implementation of the rules as the EventB2Java tool. Users can benefit from the work accomplished in this chapter since the following reasons

- the EventB2Java tool generates both sequential and multithreaded Java code,
- EventB2Java can be applied to both abstract and refinement Event-B models, and
- the generation of JML specifications enable users to write customised code that replaces the code generated by EventB2Java, and then to use existing JML tools [40, 66] to verify that the customised code is correct.

Work done in this chapter have been published in [91] and in a book chapter [36], and submitted to a journal paper [92]. My participation was to define the sound rules for the translation of Event-B models to JML-annotated Java code, and to fully implement the EventB2Java tool.

Chapter §7. Case Studies. We have validated the implementation of EventB2Java by applying it to several Event-B models. This chapter presents the set of Event-B models and the JML-annotated Java code generated by EventB2Java. This chapter also presents two case studies using EventB2Java: the first case study is on the development of an Android [61] application. This development demonstrates how EventB2Java can be used as part of a software development methodology to generate the functionality (the Model) of an Android application that is organised following the MVC (Model- View-Controller) design pattern [60]; the second case study is on testing the behaviour of an Event-B model of the Tokeneer safety critical system [47]. This development demonstrates how EventB2Java and Java Unit (JUnit) testing [73] can be used to refine (improve) an Event-B model to conform to an existing System Test Specification (STS) document. This chapter also presents a benchmark that compares the EventB2Java tool with existing tools for generating Java code from Event-B models. We compare EventB2Java against EB2J [77], Code Generation [54] tools for nine Event-B models and six comparison criteria.

Contributions. The main contributions of this chapter are *i*) the presentation of two case studies using the EventB2Java tool, *ii*) the presentation of a benchmark comparing EventB2Java against two existing tools that generates Java code from Event-B models. Work done in this chapter have been published in [92, 36]. My participation on this work was: regarding the first case study, I modelled the system in Event-B (and discharged all proof obligations). The system is an extension of an existing Event-B model of a Social Network. I also implemented the Controller of the system in Java, and implemented the View using Android API; regarding the second case study, I participated modelling the Event-B model for Tokeneer and discharging proof obligations of the model. I implemented the Java Unit test cases; I participated in the definition of the criteria for the benchmark, and I undertook the comparison of the existing tools for generating Java code from Event-B models against EventB2Java.

Chapter 2

Background

2.1 The B method

The B method [2] is a strategy for software development in which an abstract model of a system is transformed into an implementation via a series of refinement steps, whereby the behaviour of each refinement is provably consistent with respect to the behaviour of the previous step. A model M_{i+1} of a system at stage $i + 1$ is said to *refine* the model M_i at stage i. The refined model must keep a palpable behavioural relation with its abstraction. This relation is modelled through a "gluing invariant" property. Refinement steps generate proof obligations to ensure that the system works correctly. Roughly speaking, a refinement model should be such that it can replace the refined model without the user of it noticing any change.

B models are called machines, composed of a static part: variables, constant, parameters and invariants; and a dynamic part: operations, which describe how the system evolves. B machines use predicate calculus (essentially predicate logic and set theory) to model properties. Machine operations are defined using various forms of substitutions. The following explains the different forms:

Figure 2.1 shows the syntax and the semantics of substitutions in B (taken from [2]). The semantics are to be valid for any predicate R. In Figure 2.1, P, Q, and R are predicates, and S and T are substitutions of variables by expressions. $[S]R$, with S equals $x := E$, denotes the predicate resulting from the substitution of any free occurrence of variable x in R by expression E.

A *preconditioned* substitution $P \mid S$ denotes the substitution S under the operation pre-condition P. Hence, the correct behaviour of the substitution S is only ensured when it is activated in a state in which P holds. When P does not hold, $P \mid S$ is not guaranteed to verify any predicate R and a crash of the system occurs. A *guarded* substitution $P \Longrightarrow S$ executes a substitution S under the assumption P, hence if P does not hold, the substitution is able to establish any predicate R.

A *bounded choice* substitution $S [] T$ non-deterministically implements a substitution among S or T. The semantics of a bounded choice substitution ensures that either substitution is implemented it must satisfy R.

An *unbounded choice* substitution $\forall x \cdot S$ generalises a bounded choice substitution for any substitution S. Figure 2.1 presents a particular unbounded substitution that further requires x to make predicate P true. P and S both depend on x and the machine variables. *Guarded bounded* substitutions combine *bounded choice* and *guarded* substitutions. *Var Choice* is a syntactic extension to the *bounded choice* substitution.

Substitution	**Syntax**	**Definition**	**Semantics**
Preconditioned	PRE P THEN S END	$P \mid S$	$[P \mid S]R$ $\Leftrightarrow$ $P \wedge [S]R$
Guarded	SELECT P THEN S END	$P \Longrightarrow S$	$[P \Longrightarrow S](R)$ $\Leftrightarrow$ $(P \Rightarrow [S]R)$
BoundedChoice	CHOICE S OR T END	$S[]T$	$(S[]T)(R)$ $\Leftrightarrow$ $([S]R \wedge [T]R)$
UnboundedChoice	ANY x WHERE P THEN S END	$\forall x \cdot$ $(P \Longrightarrow S)$	$\forall x \cdot P \Longrightarrow [S]R$
GuardedBounded	SELECT P THEN S WHEN Q THEN T END	CHOICE $P \Longrightarrow S$ OR $Q \Longrightarrow T$ END	$(P \Longrightarrow S [] Q$ $\Longrightarrow T)(R)$ $\Leftrightarrow$ $(P \Rightarrow [S]R) \wedge$ $(Q \Rightarrow [T]R)$
VarChoice	VAR x IN S END	$\forall x \cdot$ $(true \Longrightarrow S)$	$\forall x \cdot true \Longrightarrow [S]R$

Figure 2.1: Substitutions in B

2.2 The Event-B Method

Event-B [4] models are complete developments of discrete transition systems. Event-B was introduced by J-R. Abrial, and is derived from the B method. Event-B models are composed of machines and contexts. Three basic relationships are used to structure a model. A machine **sees** a context and can **refine** another machine. And a context can **extend** another context.

Machines contain the dynamic parts of a model (e.g. variables, invariants, events). And contexts the static part of a model (e.g. carrier sets, constants). Figure 2.2 shows a general structure of an Event-B machine. It contains a list of machines and contexts the machine **refine**s and **see**s. It also contains a list of **variables** used in the machine, a list of **invariants** that restricts the possible values the variables can take, and a **variant** which is a numeric expression that has to be decreased by special events. The **events** of a machine determine the way the system evolve. The system evolves via a serie of substitutions of variables whenever an event is triggered. **events** contain a clause **status** that defines an event as **ordinary**, **convergent** (the event has to decrease the **variant**), or **anticipated** (the event must not increase the **variant**). It also contain a list of local variables (they can be seen as parameter of the event) under the clause **any**. Events contain a guard (under clause **where**) that needs to be true in order for the event to be triggered. If the guard is true, the event might perform its actions. Actions are under the clause **then** and they define how the system evolves by means of substitutions. In Event-B there are two kind of substitutions: deterministic assignment, which takes the form of <*variable_identifier*> := <*expression*>; and non-deterministic assignment, which takes the form of <*variable_identifier_list*> :| <*before_after_predicate*>, the *before_after_predicate* is the relationship that exists between the value of a variable *just before* and *just after* the assignment. It may contain machine variables. Non-deterministic assignments generalise deterministic assignments. For example, $v := v + w$ can be expressed as $v :\mid v' = v + w$, where v' is the value of v after the assignment.

<machine_identifier >
refines
< machine_identifier >
sees
< context_identifier_list >
variables
< variable_identifier_list >
invariants
< label >: < predicate >
variants
< variant >
events
< event_list >

Figure 2.2: General structure of Event-B machine (taken from [4]).

Figure 2.3 shows a general structure of an Event-B context. It contains a list of **extends** which defines which contexts this context is extending. It also contains a list of carrier sets (**sets**) and a list of **constant**s. Finally, it defines **axiom**s which sets and constants must obey.

```
<context_identifier >
  extends
    < context_identifier_list >
  sets
    < set_identifier_list >
  constants
    < constant_identifier_list >
  axioms
    < label >: < predicate >
```

Figure 2.3: General structure of Event-B context (taken from [4]).

2.3 Tool support for Event-B

The Rigorous Open Development Environment for Complex Systems (RODIN) [28] is an open-source Eclipse IDE that provides support for Event-B and that provides a set of tools for working with Event-B models, e.g. an editor, a proof generator, and provers. Existing Rodin plug-ins provide extended functionality such as model checking and animation [72].

Rodin come with an API that offers a series of Java interfaces for manipulating Event-B components called the data model. It also comes with a persistence layer (called the Rodin database) that uses `XML` files to store these components. It is intended to abstract the concrete persistence implementation from the data model. The database API is located in the `org.rodinp.core` package. Full source code for Rodin is available in [93].

2.3.1 Rodin Proof Obligations

In modelling in Event-B, users transform an abstract machine to code via a series of refinements. Refinements add new information to the system. Each refinement must be correct with the previous one. Hence a set of Proof Obligations (PO) is generated. POs are axioms that need to be proven true in oder for the underlying system to be correct. Rodin automatically generates them. Rodin provides the tool proof generator, and several provers. The provers associated to Rodin help users to discharge PO, however sometimes it is necessary the assistance of the user to discharge them. Generated POs are describe in this section.

The Rodin proof-obligation generator automatically generates proof obligations based on both the machine and the context. As explained above, there are three kind of relations between machines and contexts: *i*) a machine **sees** a context, *ii*) a concrete machine can **refine** an abstract machine, and *iii*) a context can **extend** another context. Given the abstract event evt_0 and the concrete event evt in Figure 2.4, and given an abstract and a concrete machine declaring the events respectively, Rodin generates several proof obligations to

```
$evt_0$
 any $x$ where $G(s, c, v, x)$
 then
   act $v :| BA_0(s, c, v, x, v')$
end

$evt$ refines $evt_0$
 any $y$ where $H(y, c, w)$
 then
   act $w :| BA(s, c, w, y, w')$
end
```

Figure 2.4: Events of an Event-B abstract and refinement machines

ensure that the machines are models of the same system yet at a different level of abstraction. In Figure 2.4, s and c are the sets and constants seen by the abstract and concrete machines, v is the set of abstract variables, w, which includes v, is the set of concrete variables, predicates G and H are the abstract and concrete guards, BA_0 and BA are *before-after* predicates that relate the state of variables before and after actions occur[1].

In Event-B, the symbol :| represents non-deterministic assignment. Non-deterministic assignments generalise deterministic assignments (formed with the aid of :=), e.g. the deterministic assignment $x := x + z$ can be expressed as the non-deterministic assignment $x :| x' = x + z$.

An abstract machine can declare an abstract invariant I and a concrete machine can additionally declare an invariant J (also called "gluing" invariant) that depends on the context and the local machine variables respectively. Contexts can further declare a set of theorems and axioms.

Rodin generates invariant preservation proof obligations (INV) for every abstract (concrete) event of the abstract (concrete) machine expressing that, given the axioms and theorems, the abstract (gluing) invariant, the guard of the event, and the before-after predicate, then the abstract (the concrete) invariant holds in the after state. Rodin generates a guard strengthening proof obligation (GRD) for every event expressing that the guard of the concrete event must be as least as strong as the guard of the abstract event. It generates a feasibility proof-obligation (FIS) for the action of every event stating that a solution to the before-after predicates exists. For every event merging two abstract events, Rodin generates a merging proof obligation (MRG) that ensures that the guard of the merging event is stronger than the disjunction of the guards of the abstract events. For every event, Rodin generates a simulation proof obligation (SIM) that ensures that abstract actions are correctly simulated by the concrete actions. That is, the result produced by the concrete action does not contradict the result produced by the abstract action. Rodin also generates numeric and finite proof obligations to ensure that variable declarations are well-defined

---

[1] Primed variables refer to after-states.

(WD) and sets are finite (FIN).

In Rodin, one can further declare a machine variant positive integer expression n that must be decreased by specialised events to ensure that they do not monopolise the system (NAT). Machine events can be declared anticipated (convergent), and Rodin generates proof obligations expressing that the modified variant (evaluated after executing the event action) should be (strictly) less than or equal to the variant evaluated before the event action occurs (VAR).

One can declare a *witness* of a refinement event with the aid of the with clause of Rodin. A witness expression relates bounded variables of an abstract event with bounded variables of the concrete event, e.g. one could have added the expression with $x : x' = y$ to the declaration of the concrete event *evt* in Figure 2.4, meaning that bounded variable x in the abstract event is renamed as bounded variable y in the concrete event. Rodin generates a witness proof obligation (WFIS) for every event witness expressing that a solution for the witness expression exists.

Theorems must be provable from contexts or machines (THM). In Event-B, Theorems are used to simplify complex proof-obligations.

2.4 The Java Modeling Language (JML)

JML [70, 25, 71] is an interface specification language for Java – it is designed for specifying the behaviour of Java classes, and is included directly in Java source files using special comment markers `//@` and `/*@ */`. JML's type system includes all built-in Java types and additional types representing mathematical sets, sequences, functions and relations, which are represented as JML specified Java classes in the `org.jmlspecs.models` package. Similarly, JML expressions are a superset of Java expressions, with the addition of notations such as ==> for logical implication, **`\exists`** for existential quantification, and **`\forall`** for universal quantification.

JML class specifications can include **`invariant`** clauses (assertions that must be satisfied in every visible state of the class), **`initially`** clauses (specifying conditions that the post-state of every class constructor must satisfy), and history constraints (specified with the keyword **`constraint`**, which are similar to invariants, with the additional ability to relate pre- and post-states of a method). Concrete JML specifications can be written directly over fields of the Java class, while more abstract ones can use specification-only **`model`** and **`ghost`** fields. **`ghost`** fields are not related to the concrete state of the class and can be declared **`final`**, while **`model`** fields are related to Java implementation fields via a **`represents`** clause, which acts much like a gluing invariant in Event-B refinement.

JML provides pre-post style specifications for Java methods describing software contracts [78]: if the caller of a method meets the pre-condition, the method must ensure the post-condition. JML uses keywords **`requires`** for method pre-conditions, **`ensures`** for normal method post-conditions, **`signals`** and **`exsures`** for method exceptional post-conditions, and **`assignable`** and

`modifies` for frame conditions (lists of locations whose values may change from the pre-state to the post-state of a method). An **`assignable`** clause of **`nothing`** prevents any location from being modified from the pre- to the post-state, and an **`assignable`** clause of **`everything`** allows any side- effect. Declaring a method as **`pure`** has the same effect as **`assignable \nothing`**. The pre-state is the state on method entry and the post-state is the state on method exit. A **`normal_behavior`** method specification states that if the method pre-condition holds in the pre-state of the method, then it will always terminate in a normal state, and the normal post-condition will hold in this state. A JML **`exceptional_behavior`** method specification states that if the method pre-condition holds in the pre-state of the method, then it will always terminate in an exceptional state, throwing a `java.lang.Exception`, and the corresponding exceptional post-condition will hold in this state. In an **`ensures`** or **`signals`** clause, the keyword **`\old`** is used to indicate expressions that must be evaluated in the pre-state of the method – all other expressions are evaluated in the post-state. The **`\old`** keyword can also be used in history constraints, providing a convenient way to specify (for example) that the post-state value of a field is always equal to the pre-state value, thus making the field a constant.

Figure 2.5 presents a simple example of a JML specified Java **`abstract`** class. This class defines an excerpt of a Social Network. In JML, **`model`** and **`ghost`** fields are both specification-only. That is, they exist in JML but they do not exist in the Java code. A significant difference is that **`model`** fields are an abstraction of the mutable part of an object's state and should be related to an implementation field through the JML **`represents`** clause in any concrete class that inherits from `SOCIAL_NETWORK`. While **`ghost`** fields can be used to represent immutable data. Classes `JMLEqualsSet` and `JMLEqualsToEqualsRelation` (in the `org.jmlspecs.models` package) are built-in to JML and represent mathematical sets and relations, respectively. The **`invariant`** is an assertion that all instances of the class must satisfy in all visible states, and the **`initially`** clause is an assertion that the initial values of the fields must satisfy.

The specification of the `transmit_rc` method uses two specification cases (the keyword **`also`**) - the first specifying that content `rc` is transmitted to person `pe` if `pe` and `ow` are actually person of the network, `rc` belongs to the content of the network, belongs to person `ow`, and `rc` has not been transmitted yet to person `pe`. The second specifying that attempting to transmit `rc` to a person `pe` when either `pe` or `ow` are not actually people of the network, or `rc` does not belong either to the content of the network or to person `ow`, or `rc` has been already transmitted to person `pe`, has no effect. The specification of the `transmit_rc` method demonstrates the syntax of existentially quantified assertions in JML, and the use of **`exceptional_behavior`** specification cases to specify when exceptions are to be thrown.

```
//@ model import org.jmlspecs.models.JMLEqualsSet;
//@ model import org.jmlspecs.models.JMLEqualsToEqualsRelation;

public abstract class SOCIAL_NETWORK {
  //@ public final ghost JMLEqualsSet<Integer> PERSON;
  //@ public final ghost JMLEqualsSet<Integer> RAWCONTENT;
  //@ public model JMLEqualsSet<Integer> person;
  //@ public model JMLEqualsSet<Integer> rawcontent;
  //@ public model JMLEqualsToEqualsRelation<Integer,Integer>  content;

  /*@ public invariant person.isSubset(PERSON)
      && rawcontent.isSubset(RAWCONTENT)
      && (new Relation<Integer, Integer> (person, rawcontent)).has(content)
      && content.domain().equals(person)
      && content.range().equals(rawcontent);*/

  /*@ public initially person.isEmpty() &&
        rawcontent.isEmpty() && content.isEmpty();*/

  /*@ public normal_behavior
      requires rawcontent.has(rc) && person.has(ow)
        && person.has(pe) && !ow.equals(pe)
        && !content.has(ModelUtils.maplet(pe,rc));
      assignable content;
      ensures (\exists JMLEqualsSet<Integer> prs;\old(prs.isSubset(person));
          content.equals(\old(ModelUtils.toRel(content.union(ModelUtils.
               toRel(ModelUtils.maplet(pe,rc)))).union(ModelUtils.cartesian(
               prs, ModelUtils.toSet(rc))))));
    also
      public exceptional_behavior
      requires !(rawcontent.has(rc) && person.has(ow)
        && person.has(pe) && !ow.equals(pe)
        && !content.has(ModelUtils.maplet(pe,rc)));
      assignable \nothing; signals (Exception) true;*/
  public abstract void transmit_rc(Integer rc, Integer ow,
  Integer pe);
}
```

Figure 2.5: A JML specification of a social networking class.

2.4.1 Tool support for JML

There are different techniques that work with JML specifications along with the proper tooling to avoid error-prone. The most basic technique is parsing and type-checking the JML specifications as done by the JML checker `jml`. Several tools have been developed to help users with the correct specification of JML clauses. For instance, the CHASE tool [31] checks the assignable clause of a JML-annotated Java program. It checks for every method its assignable clause by checking for every assignment and for every method call in the body, whether it agrees with the assignable clause of the method that is checked. The tool gives feed-back to the user on the forgetting variables that may be modified by a specific method. Another tool that helps users on specifying JML clauses is Daikon [57]. Daikon automatically infers invariants from JML-annotated Java programs.

Another more specialised technique that works with JML consists of checking the correctness of JML by run-time checking. In run-time checking one runs the Java code and tests some safety conditions. Such run-time checking is done by the JML compiler `jmlc` [43]. `jmlc` is an extension of the Java compiler that compiles JML-annotated Java programs into Java bytecode. The compiled bytecode contains the safety properties (assertions) as pre-conditions, normal and exceptional post-conditions, invariants, and history constraints. If an assertion is violated, an error message is arisen. Another tool for assertion checking during testing phase is `jmlunit` [105]. `jmlunit` uses the JML specification as a testing oracle, automating the process of generating JUnit tests. The tests creating by this tool catches any assertion given by the JML run-time assertion, thus it checks if, for instance, a pre-condition or an invariant is violated, meaning the Java code does not meet the JML specification.

The major problem with run-time checking is that it is limited by the execution paths done by the test suite (since it is executed in run-time). Another technique that works with JML is the static verification of the Java code. This can give more assurance in the correctness of the Java code as it establishes the correctness for all possible execution paths. Typically, this technique generates proof obligations from the JML specification and uses a theorem prover to discharge them. There are several tool for working using this technique: the LOOP tool [99] works over sequential Java implementations. It translates proof obligations and uses PVS [84] or Isabelle [88] to discharge the proof obligations; Krakota [75] is well suited for Java Cards Applets. Krakota receives as an input an JML-annotated Java program and translates it to the input language of the WHY tool [58]. WHY is used to automatically generate proof obligations and uses COQ [16] prover to discharge the proofs; ESC/JAVA2 [46] is intended to detect more simple errors, like null pointers, out of bound array access. ESC/JAVA2 uses provers like Z3 [49] to discharge proofs obligations; OpenJML [45] translates JML specifications into SMTLIB [12] (Satisfiability Modulo Theories Library) format and passes the proof problems implied to backend SMT solvers; the JACK tool [26] works on Java cards applets. The input for JACK is a JML-annotated Java program where user needs to express the proof obligation (property) that want to proof. The tool translates the program to the input language of B method, and translates the proof obligations to lemmas in B. Lemmas are proven using a prover developed within Atelier B [8]; jmle [67] translates JML specifications to Java programs that are executed using the Java Constraint Kit.

There are several more tools for checking the JML specification against the Java code. L. Burdy et. al. had made an overview of said tools in [25].

One limitation about these tools is no one has been developed to work with Java 7 that introduces generics types. A generic type is a generic class that is parameterised over types, so it is not possible to know the specific type in a static manner.

2.5 Dafny

Dafny [65] is an imperative object-based language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs. Dafny runs under Microsoft Visual Studio, and from the command line, which requires a .NET virtual machine. Dafny provides support for the annotation of program as contracts: pre- and post-conditions. Also provides support for abstract specifications through the definition and use of **`ghost`** variables and methods, and for the definition and specification of mathematical functions. Functions are specification-only constructs; they exist for verification-only purposes and are **`ghost`** by default. The **`requires`** specification of the function (which goes in the same direction of **`requires`** JML clause) may be used to define its domain (partial functions). Post-conditions are written as **`ensures`** specifications (which also goes in the same direction of **`ensures`** JML clause). It represents the post-state of the function and must hold for every possible invocation of the function. Pre- and post-conditions must be written at the beginning of the function. The **`assert`** clause can be written somewhere in the middle of the function. It tells that a particular expression always holds when control reaches that part of the code. The **`reads`** part declares the function's frame condition, which is all the memory locations that the function is allowed to read [65]. Finally, the **`decreases`** part states the termination metrics of the function.

Program verification with Dafny works by translating the program written in Dafny to the Boogie 2 proving engine [11] in such a way that the correctness of the Boogie program implies the correctness of the Dafny program.

2.6 Software Design Patterns

Code patterns are common to many software solutions. According to [7], a software pattern "creates a common structure to help software developers to resolve recurring problems encountered throughout software development".

Gamma et. al. popularised the term Software Design Patterns [60], which is a reusable solution to a common problem within a given context. A Design Pattern is not an implementation of a solution but a template for how to address the problem. Gamma describes different kind of design patterns, some examples are the following:

Singleton: ensures that a class has one instance, provides a global point of access. For instance, in the development of a game that defines a board to be modified among the players. The board needs to be instantiated once since the players are going to be in the same game.

Facade: provides a unified interface to a set of interfaces in a subsystem. For instance, a common way to reduce complexity in the development of software is to implement the system into subsystems. However, that means the subsystems need to communicate each other. In order to minimize this communication burden, and dependencies between subsystems, one can use a *facade* as a front-end.

Abstract Factory: provides an interface for creating families of related or dependent objects without specifying their concrete classes. For instance, it can be used when developing a framework that needs to be portable to different platforms.

Visitor: allows to define a new operation without changing the classes of the elements on which it operates. For instance, when defining a translation from language A to language B, one normally creates an Abstract Syntax Tree (AST) for A and then the translation to B is attached to the nodes of that tree. Once the AST is built one can also think about another feature of the development like the *pretty print* option. However, it will be messy to combine the translation with this new feature. The *Visitor* design pattern proposes a solution to add this new feature without changing the AST.

Model-View-Controller (MVC): models software as composed of three components: a model (M), a controller (C), and views (V). It separates the internal representation of the information from the way the users see it. All components interact with each other; the Controller is a bridge between the Model and the Views, sends commands to the Model for it to update its state. It also sends commands to the Views to change the presentation of the Model (this process is called rendering); the Model contains the logic of the system, sends information to the Controller and the Views every time the Model changes its state; the Views, which are the graphical representation of the information, request information from the Model necessary to update the information presented to the user.

This design pattern is commonly used for the development of Graphical User Interfaces (GUI) since the separation of the internal presentation of information allows developers to change just the Views.

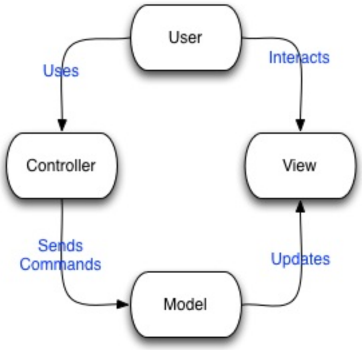


Figure 2.6: MVC design pattern.

A common interaction of the MVC components is depicted in Figure 2.6 for the development of a web calculator. The View deploys the graphical part of the calculator. The user interact with the View generating a request from the Model (e.g. $3 + 1$). The Controller intercepts the user interaction and sends a command to the Model for the Model to change its state. The Model receives the command, processes it, and communicates with the View, so View changes the representation of the information that users see (e.g. 4).

Chapter 3

Translating B Machines to JML Specifications

This chapter. This chapter presents a translation from B machines to JML specifications defined through syntactic rules. It also presents the implementation of this translation as the B2Jml tool. The tool enables B experts to use Refinement Calculus techniques to develop critical components in B, and then translate the result to JML for developers with less mathematical expertise to be able to implement code that respects the JML specification. B2Jml enables developers to use lightweight JML tools such as the jmle tool for executing JML specifications [66, 40], runtime assertion checkers and static analysers [25]. B2Jml fully supports the B syntax except for the B constructs for multiple incremental specification of machines, e.g. for including, importing, seeing, or extending other machines. We integrated B2Jml to the ABTools suite [18]. We have validated the tool by applying it to a moderately complex B model of a social networking site. We further executed the resulting JML specifications against a suite of test cases developed for a hand-translation of a B model. The full code of the implementation of the B2Jml tool is available at [20].

The work presented in this chapter inspired the work presented in the rest of this thesis. It has been published in [41, 36]. I participated in this work at the end of the implementation of the B2Jml tool.

The rest of this chapter is organised as follows. Section 3.1 describes the set of transition rules that translate B models to JML abstract class. These rules are implemented as the B2Jml tool and integrated into the ABTools suite. Section 3.2 presents the implementation. We have validated B2Jml by applying it to the B model presented in Section 3.3. The B model is a moderately complex model of a social networking site. That section also presents the resulting JML abstract Java class. Finally, Section 3.4 gives conclusions.

Contributions. *i*) We present the definition of the translation of B machines into JML specifications via syntactic rules, and *ii*) we present the implementation of this translation as B2Jml tool. Users might prefer to use JML since *a*) implementing a JML specification in Java is much more straightforward than implementing an equivalent B machine in Java. And *b*) the user may be more familiar with JML syntax than B notation.

Related work. Jin and Yang [62] outline an approach for translating VDM-SL to JML. Their motivations are similar to B2Jml tool in that they view VDM-SL as a better language for modelling at an abstract level (as in B), and JML as a better language for working closer to an implementation level. Their approach has not been automated – they describe only a strategy for translating specifications by hand.

Boulanger [19] describes partially automated translations between B and VHDL (in both directions). This approach permits co-design – verified B implementation machines can be translated to VHDL for realisation in hardware, and translations of VHDL libraries can be used by B machines. Hence, these translations allow designers to verify models of hardware components.

Bouquet et. al. have defined a translation from JML to B[22] and implemented their approach as the JML2B tool [23]. Although their translation goes in the opposite direction from the work presented here, their motivation is quite similar – they view translation as a way to gain access to more appropriate tools for the task at hand, which in this case is verifying the correctness of an abstract model without regard to code. JML verification tools are primarily concerned with verifying the correctness of code with respect to specifications, while B has much stronger tool support for verifying models.

In some ways, translating from JML to B is a more difficult problem than the reverse, as JML includes many concepts (objects, inheritance, exceptions, etc.) that do not appear in B. Hence, Bouquet et al. were required to build representations of these concepts in B for use by their translated machines. Distinguishing pre- and post-state values required considerable effort, while in B2Jml translation it was relatively clear which parts of a B machine should be evaluated in the pre-state, and which parts needed to be evaluated in the post-state. The translations are also similar because the correspondence between PRE substitutions and **`requires`** clauses; invariants in both languages; B operations and JML methods and so on is straightforward. One significant difference is that Bouquet et. al. translate a JML class specification to a B machine that has a set variable containing all instances of the class. Additional variables of the machine represent each JML field as a function from this set of instances to the value of the field for that instance. This provides a mechanism for distinguishing pre- and post-state values (by making copies of these functions), but also forces the B operation representing a JML method take the calling object as an explicit parameter, rather than referring directly to the machine variables in the usual way. This makes the correspondence between the JML specification and the B machine more difficult to see.

3.1 The Translation from B to JML

The translation from B to JML is implemented with the aid of a B2Jml operator. It is defined (via syntactic rules), it takes B syntax as input and returns the corresponding JML specification. To assist in this translation, we defined a MOD operator. It calculates the set of variables modified by an operation. The definition of MOD is inspired by the syntactic rules backing the analysis performed by the Chase tool in [31]. Further, a TypeOf operator is employed (without definition) to denote the inference of the type of a B variable and its translation into a corresponding JML type. Correspondence between B and JML types is briefly described at the end of section 3.1.2.

B2Jml translates a B abstract or refinement machine to a JML annotated abstract Java class. The machine variables become **`model`** fields, and operations are translated to abstract methods with JML specifications. A preconditioned substitution in B generates JML (normal and exceptional) method specification cases. Additionally, although substitutions in B have no explicit notion of post-condition, B2Jml translates other substitutions to JML post-conditions that relate the pre- and post-state values of the variables modified by the substitution. The rules defining B2Jml are deterministic so one cannot apply two different rules at the same time. In this sense, these rules define a calculus that computes the translation of B into JML.

Section 3.1.1 presents the translation of general substitutions, Section 3.1.2 presents the translation of other B syntax.

3.1.1 Translating Substitutions

Rule Sel translates a guarded substitution to a JML implication (==>) in which the antecedent is obtained from the translation of the guard and the consequent is obtained from the translation of the nested substitution, this matches the definition of guarded substitution in B presented in Figure 2.1 where in a guarded substitution WHEN P THEN S END, substitution S is executed under the assumption of P. Rules When generalise rule Sel. The first rule When is a synonym of rule Sel, in B, the construct for WHEN in the form WHEN P THEN S END can be seen as SELECT P THEN S END so it is translated in the same way as SELECT. The second rule When considers two guards[1].

$$\begin{array}{c}\dfrac{\mathsf{Pred}(P) = \texttt{P} \quad \mathsf{B2Jml}(S) = \texttt{S}}{\mathsf{B2Jml}(\mathsf{SELECT}\ P\ \mathsf{THEN}\ S\ \mathsf{END})}\ (\mathsf{Sel}) \\ = \\ \backslash\texttt{\textbf{old}(P) ==> S}\end{array}$$

$$\begin{array}{c}\dfrac{\mathsf{Pred}(P) = \texttt{P} \quad \mathsf{B2Jml}(S) = \texttt{S}}{\mathsf{B2Jml}(\mathsf{WHEN}\ P\ \mathsf{THEN}\ S\ \mathsf{END})}\ (\mathsf{When}) \\ = \\ \backslash\texttt{\textbf{old}(P) ==> S}\end{array}$$

---

[1] The rule for the simultaneous substitution $S \ || \ SS$ is presented later in this section.

$$\frac{\begin{array}{ll}\mathsf{Pred}(P) = \texttt{P} & \mathsf{B2Jml}(S) = \texttt{S} \\ \mathsf{Pred}(Q) = \texttt{Q} & \mathsf{B2Jml}(T) = \texttt{T}\end{array}}{\begin{array}{c}\mathsf{B2Jml(SELECT}\ P\ \mathsf{THEN}\ S \\ \mathsf{WHEN}\ Q\ \mathsf{THEN}\ T\ \mathsf{END)} \\ = \\ \texttt{(\textbackslash \textbf{old}(P) ==> S) \&\&} \\ \texttt{(\textbackslash \textbf{old}(Q) ==> T)}\end{array}}\ (\mathsf{When})$$

Rules If and IfElse translate the IF and IF ELSE substitutions to JML.

$$\frac{\mathsf{Pred}(P) = \texttt{P}\ \mathsf{B2Jml}(S) = \texttt{S}}{\begin{array}{c}\mathsf{B2Jml(IF}\ P\ \mathsf{THEN}\ S\ \mathsf{END)} \\ = \\ \texttt{\textbackslash \textbf{old}(P) ==> S}\end{array}}\ (\mathsf{If})$$

$$\frac{\mathsf{Pred}(P) = \texttt{P}\ \mathsf{B2Jml}(S) = \texttt{S}\ \mathsf{B2Jml}(T) = \texttt{T}}{\begin{array}{c}\mathsf{B2Jml(IF}\ P\ \mathsf{THEN}\ S\ \mathsf{ELSE}\ T\ \mathsf{END)} \\ = \\ \texttt{(\textbackslash \textbf{old}(P) ==> S)} \wedge \texttt{(!\textbackslash \textbf{old}(P) ==> T)}\end{array}}\ (\mathsf{IfElse})$$

Rule Pre presents the translation of preconditioned substitutions. A preconditioned substitution is conceptually different from a guarded substitution. While a guarded substitution imposes a condition on the internal behaviour of the machine, a preconditioned substitution imposes a condition (the precondition) on the caller. Hence, a preconditioned substitution aborts if the pre-condition does not hold. This matches the definition of preconditioned substitutions presented in Figure 2.1 where in order to execute the substitution S in a preconditioned substitution PRE P THEN S END one must prove P. In JML, this behaviour is modelled by throwing an exception.

$$\frac{\mathsf{Pred}(P) = \texttt{P}\ \mathsf{MOD}(S) = \texttt{A}\ \mathsf{B2Jml}(S) = \texttt{S}}{\begin{array}{l}\quad\ \mathsf{B2Jml(PRE}\ P\ \mathsf{THEN}\ S\ \mathsf{END)} = \\ \texttt{/*@\textbf{public normal\_behavior}} \\ \quad \texttt{\textbf{requires} P; \textbf{assignable} A;} \\ \quad \texttt{\textbf{ensures} S;} \\ \ \ \texttt{\textbf{also public exceptional\_behavior}} \\ \quad \texttt{\textbf{requires} !P; \textbf{assignable} \textbackslash \textbf{nothing};} \\ \quad \texttt{\textbf{signals(Exception) true}; */}\end{array}}\ (\mathsf{Pre})$$

Rule Choice below introduces the translation for bounded choice substitutions, whose meaning is the meaning of any of the nested substitutions.

$$\frac{\mathsf{B2Jml}(S) = \texttt{S}\ \mathsf{B2Jml}(T) = \texttt{T}}{\mathsf{B2Jml(CHOICE}\ S\ \mathsf{OR}\ T\ \mathsf{END)} = \ \texttt{S || T}}\ (\mathsf{Choice})$$

Rule Any generalises rule When for unbounded choice substitutions. The type of the variable x is inferred from its usage in the predicate P and substitution S. If at least one value of x satisfies P, any value can be chosen for use in S. If no x satisfies P, the substitution is equivalent to skip.

$$\frac{\mathsf{Pred}(P) = \texttt{P} \quad \mathsf{B2Jml}(S) = \texttt{S} \quad \mathsf{TypeOf}(x) = \texttt{Type}}{\begin{array}{l}\mathsf{B2Jml}(\mathsf{ANY}\ x\ \mathsf{WHERE}\ P\ \mathsf{THEN}\ S\ \mathsf{END}) = \\ (\backslash\textbf{exists}\ \texttt{Type}\ x\texttt{;}\ \backslash\textbf{old}\texttt{(P) \&\& S)} \mid\mid \\ (\backslash\textbf{forall}\ \texttt{Type}\ x\texttt{;}\ !\backslash\textbf{old}\texttt{(P))}\end{array}}\ (\mathsf{Any})$$

The VAR construct in B introduces a local variable x in the scope of a substitution S, and so is equivalent to an ANY substitution that does not constrain its bound variable [2].

$$\frac{}{\begin{array}{c}\mathsf{B2Jml}(\mathsf{VAR}\ x\ \mathsf{IN}\ S\ \mathsf{END}) \\ = \\ \mathsf{B2Jml}(\mathsf{ANY}\ x\ \mathsf{WHERE}\ \mathsf{TRUE}\ \mathsf{THEN}\ S\ \mathsf{END})\end{array}}\ (\mathsf{Loc})$$

Rule Asg presents the translation of an assignment from an expression E to a variable v, the simplest nontrivial substitution in B. This substitution is mapped to a JML predicate in which the value of the variable in the post-state equals the value of the expression evaluated in the pre-state.

$$\frac{\mathsf{Pred}(E) = \texttt{E}}{\mathsf{B2Jml}(v := E) = v\texttt{.equals(}\backslash\textbf{old}\texttt{(E))}}\ (\mathsf{Asg})$$

Rule Sim presents the rule for the simultaneous substitution $S \mid\mid SS$, in which SS could be another simultaneous substitution. As the name indicates, the nested substitutions occur simultaneously. Note that our rules translate $x := y \mid\mid y := x$ to $x\texttt{.equals(}\backslash\textbf{old}(y)\texttt{)}\ \&\&\ y\texttt{.equals(}\backslash\textbf{old}(x)\texttt{)}$, which matches the B semantics. B does not allow simultaneous assignments to the same variable.

$$\frac{\mathsf{B2Jml}(S) = \texttt{S} \quad \mathsf{B2Jml}(SS) = \texttt{SS}}{\begin{array}{c}\mathsf{B2Jml}(S \mid\mid SS) \\ = \\ \texttt{S \&\& SS}\end{array}}\ (\mathsf{Sim})$$

Further, frame conditions are checked; the only variables modified by a general substitution are those variables modified by assignments within the substitution. Hence, a set of Mod rules are defined to calculate the set of these variables. In rule ModAsg below, the assigned variable is added to the frame-condition set. Another rules for B substitutions are ModSel, ModGua, ModAny, ModCho and ModSim. These rules are similar to the ones underpinning the checking performed by the Chase tool [31].

$$\frac{}{\mathsf{MOD}(v := E) \;=\; \{v\}}\;(\mathsf{ModAsg})$$

$$\frac{\mathsf{MOD}(S) = \mathtt{S}}{\begin{array}{c}\mathsf{MOD}(\mathsf{SELECT}\ P\ \mathsf{THEN}\ S\ \mathsf{END})\\ = \\ \mathtt{S}\end{array}}\;(\mathsf{ModSel})$$

$$\frac{\mathsf{MOD}(S) = \mathtt{S}\quad \mathsf{MOD}(T) = \mathtt{T}}{\begin{array}{c}\mathsf{MOD}(\mathsf{SELECT}\ P\ \mathsf{THEN}\ S\ \mathsf{WHEN}\ Q\ \mathsf{THEN}\ T\ \mathsf{END})\\ = \\ \mathtt{S}\ \cup\ \mathtt{T}\end{array}}\;(\mathsf{ModGua})$$

$$\frac{\mathsf{MOD}(S) = \mathtt{S}}{\begin{array}{c}\mathsf{MOD}(\mathsf{ANY}\ x\ \mathsf{WHERE}\ P\ \mathsf{THEN}\ S\ \mathsf{END})\\ = \\ \mathtt{S}\end{array}}\;(\mathsf{ModAny})$$

$$\frac{\mathsf{MOD}(S) = \mathtt{S}\quad \mathsf{MOD}(T) = \mathtt{T}}{\begin{array}{c}\mathsf{MOD}(\mathsf{CHOICE}\ S\ \mathsf{OR}\ T\ \mathsf{END})\\ = \\ \mathtt{S}\ \cup\ \mathtt{T}\end{array}}\;(\mathsf{ModCho})$$

$$\frac{\mathsf{MOD}(S) = \mathtt{S}\quad \mathsf{MOD}(T) = \mathtt{T}}{\begin{array}{c}\mathsf{MOD}(S\ \|\ T)\\ = \\ \mathtt{S}\ \cup\ \mathtt{T}\end{array}}\;(\mathsf{ModSim})$$

As the variable introduced by a VAR substitution is local to that substitution, it should not appear in an **`assignable`** clause and so is removed via rule ModVar.

$$\frac{\mathsf{MOD}(S) = \mathtt{S}}{\begin{array}{c}\mathsf{MOD}(\mathsf{VAR}\ x\ \mathsf{IN}\ S\ \mathsf{END})\\ = \\ \mathtt{S}\ -\ \{x\}\end{array}}\;(\mathsf{ModVar})$$

3.1.2 Beyond Substitutions

First the translation of an entire B machine into a JML-annotated Java class is presented, followed by the translation of the components of that machine. As presented here, the translation considers only a single carrier set, only a single variable and so on, but can easily be extended to multiple carrier sets, variables, etc.

$$\dfrac{\begin{array}{ll}\mathsf{B2Jml}(\mathsf{SETS}\ s)=\mathtt{S} & \mathsf{B2Jml}(\mathsf{CONSTANTS}\ c)=\mathtt{C}\\ \mathsf{B2Jml}(\mathsf{VARIABLES}\ v)=\mathtt{V} & \mathsf{B2Jml}(\mathsf{PROPERTIES}\ P)=\mathtt{P}\\ \mathsf{B2Jml}(\mathsf{INVARIANT}\ I)=\mathtt{I} & \mathsf{B2Jml}(\mathsf{ASSERTIONS}\ A)=\mathtt{A}\\ \mathsf{B2Jml}(\mathsf{INITIALISATION}\ B)=\mathtt{B} & \mathsf{B2Jml}(Q)=\mathtt{Q}\end{array}}{\begin{array}{l}\mathsf{B2Jml}(\mathsf{MACHINE}\ M\\ \quad\mathsf{SETS}\ s\\ \quad\mathsf{CONSTANTS}\ c\\ \quad\mathsf{PROPERTIES}\ P\\ \quad\mathsf{VARIABLES}\ v\\ \quad\mathsf{INVARIANT}\ I\\ \quad\mathsf{ASSERTIONS}\ A\\ \quad\mathsf{INITIALISATION}\ B\\ \quad\mathsf{OPERATIONS}\ \mathsf{op}=Q\\ \quad\ \mathsf{END})=\\ \mathbf{public\ abstract\ class}\,M\ \{\\ \quad\mathtt{S\ C\ V\ P\ I\ A\ B}\\ \\ \quad\mathtt{Q}\\ \quad\mathbf{public\ abstract\ void}\,op\mathtt{();}\\ \}\end{array}}\ (\mathsf{M})$$

As there is not information about carrier sets, they are simply modelled as sets of integers as shown in Rule Set. All constants are being translated as **`final ghost`** variables, using **`ghost`** variables because JML **`model`** variables can not be declared **`final`**. Like **`model`** variables, **`ghost`** variables are specification only and so do not appear directly in implementations. Note that different instances of the class could use different carrier sets, so the field should not be static.

$$\dfrac{}{\begin{array}{c}\mathsf{B2Jml}(\mathsf{SETS}\ s)\\ =\\ \mathtt{/*@}\ \mathbf{public\ final\ ghost}\\ \mathtt{JMLEqualsSet{<}Integer{>}}\ s\mathtt{;\ */}\end{array}}\ (\mathsf{Set})$$

Rule Enum translates enumerated sets to sets of strings, where each string is the name of an enumeration constant.

$$\dfrac{}{\begin{array}{l}\mathsf{B2Jml}(\mathsf{VARIABLES}\ v=\{s1,\ldots,sn\})=\\ \mathtt{/*@}\ \mathbf{public\ static\ final\ ghost}\\ \qquad\mathtt{JMLEqualsSet{<}String{>}\ v}\\ \qquad\mathtt{=\ JMLEqualsSet.convertFrom(}\\ \quad\ \mathbf{new}\ \mathtt{String\ []}\ \{\text{“}s1''\text{,}\ldots\text{,}\text{“}sn''\}\mathtt{);*/}\end{array}}\ (\mathsf{Enum})$$

Rule Cons uses TypeOf to infer the type of a constant from the PROPERTIES section of the machine.

$$\frac{\mathsf{TypeOf}(c)\ =\ \mathsf{Type}}{\begin{array}{c}\mathsf{B2Jml}(\mathsf{CONSTANTS}\ c)\\ =\\ \texttt{//@}\ \texttt{\textbf{public static final ghost}}\ \mathsf{Type}\ c\texttt{;}\end{array}}\ (\mathsf{Cons})$$

As B PROPERTIES clauses specify properties of constants, Rule Prop translates them as **`static invariants`**.

$$\frac{\mathsf{Pred}(P)\ =\ \mathsf{P}}{\begin{array}{c}\mathsf{B2Jml}(\mathsf{PROPERTIES}\ P)\\ =\\ \texttt{//@}\ \texttt{\textbf{public static invariant}}\ \mathsf{P}\texttt{;}\end{array}}\ (\mathsf{Prop})$$

Ordinary machine variable declarations are translated to JML **`model`** variables. The type of the variable is inferred from the machine invariant.

$$\frac{\mathsf{TypeOf}(v)\ =\ \mathsf{Type}}{\begin{array}{c}\mathsf{B2Jml}(\mathsf{VARIABLES}\ v)\\ =\\ \texttt{//@}\ \texttt{\textbf{public model}}\ \mathsf{Type}\ v\texttt{;}\end{array}}\ (\mathsf{Var})$$

B invariant are translated to JML **`invariants`** and B assertions are translated as redundant invariants, as both assertions and redundant invariants are implied by ordinary invariants.

$$\frac{\mathsf{Pred}(I)\ =\ \mathsf{I}}{\mathsf{B2Jml}(\mathsf{INVARIANT}\ I)\ =\ \texttt{//@}\ \texttt{\textbf{public invariant}}\ \mathsf{I}\texttt{;}}\ (\mathsf{Inv})$$

$$\frac{\mathsf{Pred}(I)\ =\ \mathsf{I}}{\mathsf{B2Jml}(\mathsf{ASSERTIONS}\ I)\ =\ \texttt{//@}\ \texttt{\textbf{public invariant\_redundantly}}\ \mathsf{I}\texttt{;}}\ (\mathsf{Ass})$$

A B INITIALISATION clause is translated to a JML **`initially`** clause, as both provide initial values for variables. The assertion within the **`initially`** clause uses == rather than calling the `equals` method if the type of `v` is primitive.

$$\frac{\mathsf{Pred}(E)\ =\ \mathsf{E}}{\mathsf{B2Jml}(\mathsf{INITIALISATION}\ v\ :=\ E)\ =\ \texttt{//@}\ \texttt{\textbf{initially}}\ v\texttt{.equals(}\mathsf{E}\texttt{);}}\ (\mathsf{Init})$$

Here, the body of a B operation is assumed as its PRE substitution, so the translation of Q is defined by Rule PRE in Section 3.1.1. Also, rules for cases in which the body of a B operation is not a PRE are defined, for B operations with input parameters, and for B operations with single and multiple output parameters. The return type of the corresponding method is either the translated type of the single output parameter, or `Object []` in order to contain the values of multiple output parameters.

The language used in B expressions is essentially predicate logic and set theory. In the translation, sets, binary relations and binary functions are being presented by the JML library model classes `JMLEqualsSet`, `JMLEqualsToEqualsRelation`, and `JMLEqualsToEqualsMap` respectively. These classes test membership using the `equals` method of the class that the elements belong to, rather than the Java == operator. Several examples of rules for translating B operators on these types are presented below, where s_i's are sets and r is a relation.

$$\frac{\mathsf{Pred}(s_1) = \mathsf{s1} \ \ \mathsf{Pred}(s_2) = \mathsf{s2}}{\mathsf{B2Jml}(s_1 \subseteq s_2) = \mathsf{s1}.\mathtt{isSubset}\,(\mathsf{s2})}\ (\mathsf{Subset})$$

$$\frac{\mathsf{Pred}(x) = \mathsf{x} \ \ \mathsf{Pred}(s) = \mathsf{s}}{\mathsf{B2Jml}(x \in s) = \mathsf{s}.\mathtt{has}\,(\mathsf{x})}\ (\mathsf{Has})$$

$$\frac{\mathsf{Pred}(r) = \mathsf{r} \ \ \mathsf{Pred}(s) = \mathsf{s}}{\mathsf{B2Jml}(r[s]) = \mathsf{r}.\mathtt{image}\,(\mathsf{s})}\ (\mathsf{Image})$$

TypeOf maps a B set type to the JML model class `JMLEqualsSet`, a relation type to `JMLEqualsToEqualsRelation`, and a function type to `JMLEqualsToEqualsMap`. As the types of B variables are specified implicitly (by stating membership in some possibly deferred set), the type must be inferred from its usage within the machine. This type inference was already implemented in ABTools (see Section 3.2), so in the implementation is translated from the representation of B types used by ABTools to the corresponding JML types. A library code to capture additional properties of B types is being used. For instance, given the B expression:

$$d \in \mathbb{P}(\mathbb{N}) \,\&\, r \in \mathbb{P}(\mathbb{N}) \,\&\, f \in d \rightarrow r$$

which states that f is a total function from d to r, the type of f is translated as `JMLEqualsToEqualsMap<Integer, Integer>` and the following is generated as part of the class invariant:

```
(new Total<Integer, Integer>(d, r)).has(f)
```

Library class `org.jmlspecs.b2jml.util.Total` represents the set of all total functions from the specified domain to the specified range, so the `has` method returns true if and only if f is a total function from d to r.

3.2 The B2Jml Tool

The B2Jml tool is integrated into ABTools [18], which is an open source environment for developing B language processing tools. Full source code for ABTools and B2Jml is available in [20]. ABTools uses ANTLR [87] to generate a parser for B. The parser constructs abstract syntax trees, which are then traversed (using an ANTLR tree walker) to infer and attach type information to each node. ABTools can currently generate refinement proof obligations and translate B machines to ASCII, LaTeX, HTML and XML formats, and has some initial support for generating C and Java implementations. This functionality is also implemented via ANTLR tree walkers.

The bulk of the B2Jml implementation is realised as an additional ANTLR tree walker, which implements the B2Jml, MOD, and TypeOf operators presented previously. The tree walker traverses the syntax tree constructed by ABTools to generate the JML specification as indicated by the rules for B2Jml, collecting the variables that are modified by each operation as a side effect. Additional utility classes implement the B operators on functions, relations and sequences that do not directly correspond to methods of the JML model classes, as well as providing support for B typing via classes such as `org.jmlspecs.b2jml.-util.Total` as previously described.

Installing the B2Jml tool: B2Jml is part of the ABTools distribution, so to use it one needs to install ABTools from eclipse. It can be installed in Eclipse downloading the sources from the SVN repository `https://abtools.svn.sourceforge.net/svnroot/abtools`. To run B2Jml one needs to add the argument `-toJML`. More detailed instructions on how to install and use the tool can be found at [33].

3.3 Using the B2Jml Tool

We have validated B2Jml tool by applying it to a moderate complex B model of a Social Networking. The model was taking from [38].

3.3.1 An Example in B

The B model presented in this section is a model of a social networking site taken from [38] that models social network content, social network friendship relations, and privacy on contents. Figure 3.1 presents a simplified version of the B method. Machine *SOCIAL_NETWORK* declares two sets, PERSON and RAWCONTENT, representing the set of all possible persons and the set of all possible content (text, video, photographs, etc.) in a social network respectively. Variables *person* and *rawcontent* are the sets of all persons and content that are actually in the network, and *content* is a relation mapping people to their own content.

```
MACHINE $SOCIAL\_NETWORK$
 SETS
   PERSON;
   RAWCONTENT

 VARIABLES
   $person, rawcontent, content$

 INVARIANT
   $persons \subseteq$ PERSON $\wedge$
   $rawcontent \subseteq$ RAWCONTENT $\wedge$
   $content \in person \leftrightarrow rawcontent \wedge$
   $dom(content) = person \wedge$
   $ran(content) = rawcontent$

 INITIALISATION
   $person := \varnothing \parallel$
   $rawcontent := \varnothing \parallel$
   $content := \varnothing$

 OPERATIONS
   $transmit\_rc(ow, rc, pe) \mathrel{\hat=}$
     PRE
       $rc \in rawcontent \wedge$
       $ow \in person \wedge$
       $pe \in person \wedge$
       $ow \neq pe \wedge$
       $pe \mapsto rc \notin content$
     THEN
       ANY$prs$
       WHERE
         $prs \subseteq person$
       THEN
         $content := content \cup \{pe \mapsto rc\} \cup prs \times \{rc\}$
       END
   END
 END
```

Figure 3.1: A B-machine for social networking.

A common operation in social networking sites is sharing content to people in the social networking. The B example is modelling this by transmitting raw content to a set of persons in the social networking (see operation *transmit_rc*). The operation is publishing a raw content *rc* (e.g. a photo) from the page of *ow* (i.e. the owner of *rc*) in the page of *pe*. If *transmit_rc* is invoked when its pre-condition (following PRE) is true, the meaning of the operation is the meaning of its substitution (the code following THEN). The operation is not guaranteed to achieve any result if invoked when its pre-condition is false.

In the definition of *transmit_rc*, $pe \mapsto rc$ represents the pair of elements (pe, rc), so that the content *rc* is explicitly transmitted to person *pe*. The construct ANY models unbounded choice substitution; it gives the implementer the opportunity to choose any value for the bound variable *prs* that satisfies the WHERE condition $prs \subseteq person$. This gives a refining or implementation machine the flexibility to additionally transmit the content *rc* to all of an as yet unspecified set of people.

3.3.2 Generating JML-annotated Abstract Java Classes

We used B2Jml tool to translate the most abstract B machine for the social networking site described in [38]. Then, the resulting JML-annotated Java abstract class was typed and syntax-checked with OpenJML [45]. Figure 3.2 presents the output of applying the B2Jml tool to the B model in Figure 3.1. Figure 3.2 shows how B2Jml tool translates B carrier sets as **`final ghost`** variables with type `JMLEqualsSet<Integer>`, and B variables as **`model`** variables and the type is calculated from the B **`invariant`** using the TypeOf operator previously introduced. The B invariant is translated as a JML **`invariant`**. The JML **`invariant`** is an assertion that all instances of the class must satisfy in all visible states, and the *initialisation* in B is translated as **`initially`** clause in JML which is an assertion that the initial values of the fields must satisfy.

In the specification of the *transmit_rc* method, the **`normal_behavior`** case guarantees that if the **`requires`** clause (pre-condition) is satisfied, no exception will be thrown, only the locations listed in the **`assignable`** clause can be modified by the method, and the post-state will satisfy the **`ensures`** clause (post-condition). In an **`ensures`** clause, expressions in **`\old`** are evaluated in the pre-state, while all other expressions are evaluated in the post-state. The **`exceptional_behavior`** case specifies that the method will throw an exception and no locations will be modified if its pre-condition is satisfied.

As a further validation step, the translated specification was executed using the jmle tool [66, 40]. This tool translates JML specifications to constraint programs, which can then be run using the Java Constraint Kit (JCK) [1]. Methods in the generated constraint programs can be called from ordinary Java code, so the programs can be used directly as (large and slow) Java implementations of the JML specifications they were generated from. As the translation rules were being developed for B2Jml, they were used to produce a hand-translation of the social networking machine. jmle was used to execute this hand-translation against a suite of JUnit test cases designed to check that the behaviour of this

```
import org.jmlspecs.models.*;

public abstract class SOCIAL_NETWORK{
 //@ public final ghost JMLEqualsSet<Integer> PERSON;
 //@ public final ghost  JMLEqualsSet<Integer> RAWCONTENT;
 //@ public model  JMLEqualsSet<Integer> person;
 //@ public model JMLEqualsSet<Integer> rawcontent;
 //@ public model JMLEqualsToEqualsRelation<Integer,Integer> content;

 /*@ public invariant person.isSubset(PERSON)
      && rawcontent.isSubset(RAWCONTENT)
      && new Relation<Integer, Integer>(person, rawcontent)).has(content)
      && content.domain().equals(person)
      && content.range().equals(rawcontent);*/

 /*@ public initially person.isEmpty() &&
   rawcontent.isEmpty() && content.isEmpty();*/

  /*@ public normal_behavior
    requires rawcontent.has(rc) && person.has(ow)
      && person.has(pe) && !ow.equals(pe)
      && !content.has(ModelUtils.maplet(pe,rc));
    assignable content;
    ensures (\exists JMLEqualsSet<Integer> prs;
    \old(prs.isSubset(person));
      content.equals(\old(ModelUtils.toRel(
            content.union(ModelUtils.toRel(
            ModelUtils.maplet(pe,rc)))).union(ModelUtils.cartesian(prs,
            ModelUtils.toSet(rc))))));
   also public exceptional_behavior
    requires !(rawcontent.has(rc) && person.has(ow)
      && person.has(pe) && !ow.equals(pe)
      && !content.has(ModelUtils.maplet(pe,rc)));
    assignable \nothing; signals(Exception) true;*/
    public abstract void transmit_rc(Integer rc, Integer ow, Integer pe);
}
```

Figure 3.2: A JML specification of a social networking class.

translation was as it was expected. This also provided a convenient way to check that B2Jml behaved as expected - when the implementation was mature enough, B2Jml was used to translate the B machine to JML, and then used jmle to translate the JML specification to a constraint program. The suite of JUnit test cases were ran against this program, confirming that the behaviour of the specification generated by B2Jml matched the behaviour of the hand translation for this set of test cases.

Finally, as all operations of the B model have been verified to preserve the machine invariant using Atelier B [8], the tool gives the confidence that all methods in the JML specification preserve the invariant as well. Note that the meanings of B machine invariants and JML class invariants are closely related - both are assertions that the machine variables/class fields must satisfy both before and after the execution of any operation/public method.

3.4 Conclusion

In this chapter we presented some translation rules to produce JML specifications from B machines. We also introduced the implementation of the rules as the B2Jml tool which is integrated to the ABTools suite. We validated B2Jml by applying it to a social networking model written in B. The B model is composed of an abstract machine that defines the core of a social networking and five refinements that add functionality to it. B2Jml was able to generate JML specifications from the fifth refinement of the Social Networking B model. As a further validation of B2Jml we used OpenJML [45] to type-check the JML specifications generated for the Social Networking B model. OpenJML uncovered some problems with our tool regarding type inference of variables, we used OpenJML's feed-back to correct these problems.

We believe software developers might find B2Jml useful in the development of correct applications. B2Jml enables users to model systems in B, where the model can be proven correct w.r.t. some properties. Then, the user decides the level of abstraction in B so as to generate JML specifications of the model. From the JML specification, the user can manually write Java code and use JML machinery, such as OpenJML [45] or jmle [66], to verify if the manually written Java code meets the JML specification. B2Jml bridges Refinement Calculus with B and Design-by-Contract with JML making the process of software development easy. B2Jml allows people from different backgrounds to use formal, as with B, and less-formal, as with JML, techniques together in the development of software. B2Jml makes easy the process of writing Java code from a B machine since having the translation in JML the Java implementation is more straightforward rather than directly implement Java code from the B machine, or refine the B machine close to an implementation machine. Besides, software developers are more familiar with the notation underlying JML than with the one in B.

The work presented in this chapter has some limitations: B2Jml does not fully support the syntax underlying B. This imposes restrictions to the user to translate B models to JML; the translation has not been proven correct in the logic of a prover. We have validated B2Jml by applying it to a B model, however, in order to gain full confidence of the tool we need to prove that the translation rules are correct in the logic of a prover; and B2Jml translates the type of B variables to JML as instances of class `org.jmlspecs.b2jml.*`. The classes defined in `org.jmlspecs.b2jml.*` are defined as generic Java classes. The issue is there is not (yet) any JML tool that allows users to verify the Java code against JML specifications when the code contains generic classes.

We decided not to maintain this tool any longer since we realised that Event-B offers more benefits (discussed later on) than B. We decided to put our effort on defining and implementing a tool that works over Event-B (as explained in Chapter 4). Event-B is in some ways a simpler language than B and so it is easier to translate. In particular, many of the constructs for multiple incremental specification of machines (such as the INCLUDES, IMPORTS, and SEES keywords) that we have not yet implemented in B2Jml are not included in Event-B, so we would not need to define translation rules for machine composition. Given the event-driven nature of Event-B, this work requires us to develop both a translation of events to JML specifications and a Java framework that models the firing behaviour of events.

Chapter 4

Translating Event-B Machines to JML Specifications

This chapter. The work presented in this chapter goes in the same direction as the one described in the previous chapter. The previous chapter showed how a B machine is translated to JML specifications. This chapter shows a translation of Event-B machine to JML. Event-B method is a derivative formalism of the B method, also introduced by Abrial J.-R. in [4].

We decided to translate Event-B machines rather than B machines firstly, because Event-B is in some ways a stronger language than B. For instance, Event-B enables users to define new events that refine the **skip** event, whereas this is disallowed in classical B. Second, an event in Event-B can be refined as several events whereas this it not possible in B. Third, the B method is devoted to the development of Correctness-by-Construction software, while the purpose of Event-B is used to model full systems (including hardware, software and environment). Fourth, the syntax of Event-B is simpler than the syntax of B so it is easier to learn, and easier to translate to another language. For instance, B is composed of two different and complementary languages, namely, the modelling language (e.g. syntax like WHEN P THEN S END) and the implementation language (e.g. syntax like IF P THEN S END), whilst Event-B is composed just of a modelling language. Many of the constructs for multiple incremental specification of machines (such as the INCLUDES, IMPORTS, and SEES keywords) that we did not implement in B2Jml tool are not included in Event-B, so their translation rules do not need to be defined for machine composition. Fourth, machine and event refinement conditions are neater and easier to handle in Event-B.

This chapter presents the definition rules for a translation from Event-B to JML specifications and the implementation of this translation as the EventB2Jml tool [34], which is a plug-in for the Rodin platform. The translation has been validated by applying the EventB2Jml tool to a moderately complex Event-B model, MIO. The MIO model is presented in [37]. The tool generated a JML-annotated Java abstract class. We further generated (manually) Java code for the abstract class generated by the tool. The rest of this chapter is organised as follows. Section 4.1 presents our approach to the translation of Event-B models into JML specifications. Section 4.2 presents the EventB2Jml tool that implements the translation, and Section 4.3 shows an example on the use of out tool. Finally, Section 4.4 gives conclusions.

Contributions. The main contributions of this chapter are *i*) The definition of a translation of Event-B models to JML through a collection of rules, one for each component of an Event-B model. *ii*) The implementation of this translation as the EventB2Jml tool, allowing users to specify an abstract system in Event-B and to concretely design it and implement it in JML.

Related work. In [77], Méry and Singh present the EB2ALL tool-set that includes the EB2C, EB2C++, EB2J, and EB2C$^\sharp$ plug-ins, each translating Event-B machines to the indicated language. Unlike EventB2Java, EB2ALL supports only a small subset of Event-B's syntax, and users are required to write a final Event-B implementation refinement in the syntax supported by the tool. In [83], Ostroumov and Tsiopoulos present the EHDL prototype tool that generates VHDL code from Event-B models. The tool supports a reduced subset of Event-B's syntax and users are required to extend the Event-B model before it can be translated. In [103], Wright defines a B2C extension to the Rodin platform that translates Event-B models to C code. The Code Generation tool [54] generates concurrent Java and Ada programs for a *tasking* extension [55] of Event-B. As part of the process of generating code with the Code Generation tool, the model need to be in a concrete refinement, and users are asked to model the flow of the execution of events in the tasking extension. EventB2Jml differs from all of these tools in that EventB2Jml does not require user intervention before translation, and can translate a much larger subset of Event-B syntax.

Jin and Yang [62] outline an approach for translating VDM-SL [63] to JML. Their motivations are similar to ours in that they view VDM-SL as a better language for modelling at an abstract level (much the way that we view Event-B), and JML as a better language for working closer to the implementation level. In fact, they translate VDM variables to Java fields, thus dictating the fields of an implementation.

4.1 The Translation from Event-B to JML

The translation from Event-B to JML is implemented with the aid of an EB2Jml operator, which translates an Event-B machine and any context that it "sees" to a JML annotated Java abstract class. Operator EB2Jml uses three helper operators (introduced later on): Pred that takes any predicate or expression in Event-B and translate it to the JML counterpart. MOD collects the set of

```
EB2Jml(sets s) = S
EB2Jml(constants c) = C
EB2Jml(axioms X(s, c)) = X
EB2Jml(theorems T(s, c)) = T
EB2Jml(variables v) = V
EB2Jml(invariants I(s, c, v)) = I
EB2Jml(events e) = E
---------------------------------------- (M)
EB2Jml(
   context ctx
    sets s
    constants c
    axioms X(s, c)
    theorems T(s, c)
   end

   machine M sees ctx
    variables v
    invariants I(s, c, v)
    events e
   end) =
public abstract class M{
   S C X T V I E
}
```

Figure 4.1: The translation of the Event-B machine M, and the context *ctx* that M sees to JML-annotated Java abstract class.

variables to be assigned in an Event-B event. And, TypeOf that translates the type of a variable in Event-B to the corresponding JML variable type.

Figure 4.1 presents Rule M, which translates a machine M that sees context *ctx*. All Event-B proof obligations are assumed to be discharged before a machine is translated, so that proof obligations and closely associated Event-B constructs (namely, witnesses and variants) need not be considered in the translation. A witness contains the value of a disappearing abstract event variable, and a variant is an expression that should be decreased by all convergent Event-B events and should not be incremented by any anticipated Event-B events. An Event-B machine is translated to a single JML-annotated Java abstract class, which can then be extended by a subclass that implements the abstract methods. This allows the translation to be re-run when the Event-B model is updated without risk of losing hand-written or generated Java code [1]. The translation of the context *ctx* is incorporated into the translation of machines that "see" the context.

---

[1] Rule defined in Chapter 5 for Event-B machines does not translate an Event-B machine to an abstract Java class but to a Java concrete class since rules in Chapter 5 generate an actual implementation of an Event-B machine.

In JML, **model** fields are an abstraction of the mutable part of an object's state, and so are appropriate for representing machine variables. Carrier sets and constants are also translated as **model** fields with the addition of a history **constraint** that prevents any change in the value of the field. **model** fields can be attached to a **represents** clause to the declaration of the associated implementation field[2]. As there is no type information about carrier sets in Event-B, they are simply translated as sets of integers as depicted by rule Set.

$$\frac{}{\begin{array}{c}\mathsf{EB2Jml}(\mathsf{sets}\ s) = \\ \texttt{/*@ }\textbf{public model}\ \texttt{BSet<Integer>}\ s\texttt{;} \\ \quad\textbf{public constraint}\ s\texttt{.equals(}\backslash\textbf{old}\texttt{(}s\texttt{)); */}\end{array}}\ (\mathsf{Set})$$

Translation of constants and machine variables is similar, except that constants are constrained to be immutable as previously described. If the constant c is of a primitive type, the translation will use == rather than the `equals` method. The helper operator TypeOf translates the type of an Event-B variable or constant to the JML representation of that type.

$$\frac{\mathsf{TypeOf}(c) = \texttt{Type}}{\begin{array}{c}\mathsf{EB2Jml}(\mathsf{constants}\ c) = \\ \texttt{/*@ }\textbf{public model}\ \texttt{Type}\ c\texttt{;} \\ \textbf{public constraint}\ c\texttt{.equals(}\backslash\textbf{old}\texttt{(}c\texttt{)); */}\end{array}}\ (\mathsf{Cons})$$

$$\frac{\mathsf{TypeOf}(v) = \texttt{Type}}{\begin{array}{c}\mathsf{EB2Jml}(\mathsf{variables}\ v) = \\ \texttt{//@ }\textbf{public model}\ \texttt{Type}\ v\texttt{;}\end{array}}\ (\mathsf{Var})$$

As axioms are often used to specify properties of constants, they are translated as invariants. In Event-B, theorems should be provable from axioms, matching the semantics of the **invariant_redundantly** clause in JML[3]. Event-B invariants are naturally translated as JML **invariants**. Operator Pred translates an Event-B predicate or expression to its JML counterpart.

$$\frac{\mathsf{Pred}(X(s,c)) = \texttt{X}}{\begin{array}{c}\mathsf{EB2Jml}(\mathsf{axioms}\ X(s,c)) = \\ \texttt{//@ }\textbf{public invariant}\ \texttt{X;}\end{array}}\ (\mathsf{Axiom})$$

$$\frac{\mathsf{Pred}(\ T(s,c)) = \texttt{T}}{\begin{array}{c}\mathsf{EB2Jml}(\mathsf{theorems}\ T(s,c)) = \\ \texttt{//@ }\textbf{public invariant\_redundantly}\ \texttt{T;}\end{array}}\ (\mathsf{Theorem})$$

---

[2] Carrier sets, constants and variables in Chapter 5 are translated as concrete Java (**static final**) fields, so no **model** field specification is then generated.

[3] Rules defined in Chapter 5 for Event-B axioms and theorems are defined as **static** making clearer that they should refer just to carrier sets and constants.

$$\frac{\mathsf{Pred}(I(s,c,v)) = \texttt{I}}{\begin{array}{l}\mathsf{EB2Jml}(\mathsf{invariants}\ I(s,c,v)) = \\ \texttt{//@ }\textbf{\texttt{public invariant}}\texttt{ I;}\end{array}}\ (\mathsf{Inv})$$

An *initialisation* event executes once to initialise the machine variables, and so is naturally translated to a JML **`initially`** clause. EB2Jml is used recursively to translate the actions of the *initialisation* event.

$$\frac{\mathsf{EB2Jml}(A(s,c,v)) = \texttt{A}}{\begin{array}{c}\mathsf{EB2Jml}(\mathsf{events}\ \mathit{initialisation}\ \mathsf{then}\ A(s,c,v)\ \mathsf{end}) = \\ \texttt{//@ }\textbf{\texttt{public initially}}\texttt{ A;}\end{array}}\ (\mathsf{Init})$$

Each other event is translated to two JML methods: a `guard` method that tests if the guard of the corresponding event holds, and a `run` method that models the execution of the corresponding event. In Rule Any below, variables bound by an any construct are existentially quantified in the translation, as any values for those variables that satisfy the guards can be chosen. The translation of the guard is included in the post-condition of the `run` method in order to bind these same variables, as they can be used in the body of the event. Translation of an event defined using a when construct (Rule When) is simpler as no variables need to be bounded. Translation of events uses an additional helper operator MOD, that calculates the set of variables assigned by the actions of an event (the JML **`assignable`** clause). Rules Any and When defined in Chapter 5 vary from the ones presented here. Rules in Chapter 5 translate an Event-B method to a Java class that extends Java Thread so to simulate the execution of the system as Event-B does. We defined the variables bound by an any construct as parameter of the methods `guard` and `run` since is more natural to treat them as parameters. Thus the JML spec does not define an quantifier existential.

$$\frac{\begin{array}{ll}\mathsf{TypeOf}(x) = \texttt{Type} & \mathsf{Pred}(G(s,c,v,x)) = \texttt{G} \\ \mathsf{MOD}(A(s,c,v,x)) = \texttt{D} & \mathsf{EB2Jml}(A(s,c,v,x)) = \texttt{A}\end{array}}{\begin{array}{r}\mathsf{EB2Jml}(\mathsf{event}\ \mathit{evt}\ \mathsf{any}\ x\ \mathsf{where}\ G(s,c,v,x) \\ \mathsf{then}\ A(s,c,v,x)\ \mathsf{end}) =\end{array}}\ (\mathsf{Any})$$

```
/*@ requires true;
    assignable \nothing;
    ensures \result <==> (\exists Type x; G); */
public abstract boolean guard_evt();

/*@ requires guard_evt();
    assignable D;
    ensures (\exists Type x; \old(G) && A);
  also
    requires !guard_evt();
    assignable \nothing;
    ensures true; */
public abstract void run_evt();
```

$$\frac{\mathsf{Pred}(G(s,c,v)) = \mathtt{G} \quad \mathsf{MOD}(A(s,c,v)) = \mathtt{D} \quad \mathsf{EB2Jml}(A(s,c,v)) = \mathtt{A}}{\mathsf{EB2Jml}(\mathsf{event}\ evt\ \mathsf{when}\ G(s,c,v)\ \mathsf{then}\ A(s,c,v)\ \mathsf{end}) =}\ (\mathsf{When})$$

```
/*@ requires true;
    assignable \nothing;
    ensures \result <==> G; */
public abstract boolean guard_evt();

/*@ requires guard_evt();
    assignable D;
    ensures A && \old(G);
  also
    requires !guard_evt();
    assignable \nothing;
    ensures true; */
public abstract void run_evt();
```

The JML specification of each `run` method uses two specification cases. In the first case, the translation of the guard is satisfied and the post-state of the method must satisfy the translation of the actions. In the second case, the translation of the guard is not satisfied, and the method is not allowed to modify any fields, ensuring that the post-state is the same as the pre-state. This matches the semantics of Event-B – if the guard of an event is not satisfied, the event cannot execute and hence cannot modify the system state.

The translation of ordinary and non-deterministic assignments via operator EB2Jml is presented below. The symbol $:\mid$ represents non-deterministic assignment. Non-deterministic assignments generalise deterministic assignments (formed with the aid of :=), e.g. $v := v + w$ can be expressed as $v :\mid v' = v + w$. The translation does not generate the `\old` operators shown below when translating an *initialisation* event to an **`initially`** clause. If variable v is of a primitive type, the translation will use == rather than the `equals` method.

$$\frac{\mathsf{Pred}(E(s,c,v)) = \mathtt{E}}{\mathsf{EB2Jml}(v := E) = v\mathtt{.equals(\backslash old(E))}}\ (\mathsf{Asg})$$

$$\frac{\mathsf{Pred}(P(s,c,v,v')) = \mathtt{P} \quad \mathsf{TypeOf}(v) = \mathtt{Type}}{\begin{array}{c}\mathsf{EB2Jml}(v :\mid P) = \\ \mathtt{(\backslash exists\ Type}\ v'\mathtt{;\ \backslash old(P)\ \&\&}\ v\mathtt{.equals(}v'\mathtt{))}\end{array}}\ (\mathsf{NAsg})$$

Multiple actions in the body of an event are translated individually and the results are conjoined. For example, a pair of actions:

act1 $x := y$
act2 $y := x$

is translated to `x == \old(y) && y == \old(x)` for integer variables x and y, which correctly models simultaneous actions as required by the semantics of Event-B.

4.1.1 Additional Translation Operators

The MOD operator *collects* the variables assigned by Event-B actions. The cases of MOD for assignments are shown below. For the body of an event, MOD is calculated by conjoining the variables assigned by all contained actions.

$$\mathsf{MOD}(v := E) = \{v\} \quad \mathsf{MOD}(v :| P) = \{v\}$$

The Pred operator translates Event-B predicates, boolean, relational and arithmetic expressions in the natural way. While some operations on Event-B sets, functions and relations have direct counterparts in the model classes `JMLEqualsSet` and `JMLEqualsToEqualsRelation` that are built-in to JML(as shown in Chapter 3), many other operations do not. To supply these operations, an implemented (and specified in JML) model classes is presented (see Section 5.3.2) `BSet` (as a subclass of `JMLEqualsSet`) and `BRelation` (as a subclass of `BSet`, a `BSet` of pairs). Note that an Event-B relation can be used anywhere that a set can appear (a relation is a set of pairs), but unfortunately `JMLEqualsToEqualsRelation` is not a subclass of `JMLEqualsSet`. Several of the rules defining Pred that translate applications of Event-B operators to calls of the corresponding methods of classes `BSet` and `BRelation` are presented below. In these rules, the s_i's are sets and r is a relation.

$$\frac{\mathsf{Pred}(s_1) = \texttt{s1} \quad \mathsf{Pred}(s_2) = \texttt{s2}}{\mathsf{Pred}(s_1 \subseteq s_2) = \texttt{s1.isSubset(s2)}}\ (\mathsf{Subset})$$

$$\frac{\mathsf{Pred}(x) = \texttt{x} \quad \mathsf{Pred}(s) = \texttt{s}}{\mathsf{Pred}(x \in s) = \texttt{s.has(x)}}\ (\mathsf{Has})$$

$$\frac{\mathsf{Pred}(r) = \texttt{r} \quad \mathsf{Pred}(s) = \texttt{s}}{\mathsf{Pred}(r(s)) = \texttt{r.image(s)}}\ (\mathsf{Image})$$

Particular types of Event-B relations (total relations, functions, etc.) are translated as `BRelations` with appropriate restrictions in the invariant as explained in Chapter 3.

The TypeOf operator translates the type of Event-B variables and constants given by Rodin to the corresponding JML type. All integral types are translated as type `Integer`, all relations and functions are translated as type `BRelation`, and all other sets are translated as type `BSet` (Section 5.3.2 explains the implementation of `BSet` and `BRelation`).

```
public class Framework {
    public static void main(String[] args) {
        M m_impl = new M_Impl();
        int n = /* the number of events in M */;
        java.util.Random r = new java.util.Random();
        while (m_impl.guard_evt_1() || m_impl.guard_evt_2()
               || ...  || m_impl.guard_evt_n()) {
            switch (r.nextInt(n)) {
                case 0 : if (m_impl.guard_evt_1())
                             m_impl.run_evt_1(); break;
                ...
                case n - 1 : if (m_impl.guard_evt_n())
                                 m_impl.run_evt_n(); break;
            }
        }
    }
}
```

Figure 4.2: A framework for executing JML-annotated Java classes translated from Event-B machines.

4.1.2 A Java Framework for Event-B

An Event-B machine continues to operate until no event can be executed – in particular, a machine can run indefinitely if the guard of at least one event always holds[4]. Any event with a satisfied guard can be executed, and all event executions are atomic. Class `Framework` of Figure 4.2 presents a typical scheduler implementation of this behaviour, assuming that class `m_impl` extends the abstract class resulting from the translation of an Event-B machine M, that overrides all abstract methods of class `M` appropriately, and that the events of machine M are $evt\_1, evt\_2, \ldots, evt\_n$. Note that the result of the translation is a JML-annotated Java abstract class that must extended by a non-abstract class (`m_impl` in this case) before the methods can be executed.

4.2 The EventB2Jml Tool

The EventB2Jml tool is implemented as a Rodin plug-in. It uses the recommended interfaces [94] to traverse the statically checked internal database of Rodin. EventB2Jml was developed in Java and has been tested on version 2.8 of Rodin.

EventB2Jml uses the Rodin API to collect all components of the machine to be translated (i.e. carrier sets, constants, axioms, variables, invariants and events) as well as the necessary information (such as the gluing invariant) from the refined machines. All this information is stored in the Rodin database and can be accessed using the `org.eventb.core` library. Event-B expressions and statements are parsed and stored as abstract syntax trees, which can be accessed and traversed using the AST library in the `org.eventb.core.ast` package [100]. The AST library provides services such as parsing a mathe-

---

[4] variants are not considering in this discussion.

matical formula; that is, computing its abstract syntax tree from a string of characters (typically entered by the end-user), and navigating through formulas implementing the `Visitor` design pattern.

The EventB2Jml implementation uses the visitor design pattern provided by this package to traverse the abstract syntax tree, generating JML specifications as described in Section 4.1. As Event-B includes set-theoretical notations that are not built in to JML, additional utility classes have been developed implementing the Event-B operators on sets and relations as JML-annotated Java classes. This allows users to generate JML specifications from any stage of the Event-B system (i.e. from the most abstract machine to any refinement). The generated specifications include the imports of the library classes just described (in the `poporo.models.JML` package), and is written to a file with the same name as the machine being translated (with `.java` extension). Full source code for EventB2Jml is available at [34].

Installing EventB2Jml tool: To work with the plug-in, one must download Rodin from `http://sourceforge.net/projects/rodin-b-sharp/` (EventB2Jml has been tested in Rodin version 2.8). Then one needs to add EventB2Jml update site to the list of the update-sites in Rodin [5]. More detailed instruction on how to install and use the tool can be found at [34]. This tool is not longer maintained since we updated the translation to include Java code (see Chapter 5).

4.3 Using the EventB2Jml Tool

We have validated EventB2Jml tool by applying it to a moderate complex Event-B model, the MIO model [37]. Subsection 4.3.1 explains the Event-B model and Subsection 4.3.2 shows the output obtained after applying EventB2Jml to the Event-B model. This subsection also shows an excerpt of the implementation of the JML-annotated Java abstract class generated by the tool.

4.3.1 An Example in Event-B

The MIO is an Event-B model of a transportation system that includes articulated buses following the main corridor routes of a city [37]. The transportation system is complemented with feeding buses connecting the city with its outskirts. A partial Event-B model of the MIO is depicted in Figures 4.3 and 4.4 (abstract and first refinement machines). The abstract machine (see left Figure 4.3) models the number of parked buses through the variable *parked*, and defines an invariant $parked \in 0\,..\,min(n, m)$ that must hold before and after all the machine events. At this refinement stage, constants n and m are abstractions of the number of buses and stations of the system. These constants are defined in the machine context *ctx* (see Figure 4.3 right).

---

[5] EventB2Jml update site: `http://poporo.uma.pt/Projects/EventB2JmlUpdate`

```
machine abstract sees ctx1
 variables
  parked
 invariant
  inv1 parked ∈ 0 .. min(n, m)
 events
  initialisation
   then
    act1 parked := 0
   end
  leave
   when
    grd1 parked > 0
   then
    act1 parked := parked − 1
  end
end
```

```
context ctx1

contants n m

axioms
  ax1 n ∈ ℕ1
  ax2 m ∈ ℕ1
end
```

Figure 4.3: An excerpt of the abstract Event-B machine for MIO (left: machine *abstract*. Right: context *ctx*1 that *M* see).

The abstract machine further defines the event *leave* that models when a bus leaves any station. At this stage, the model does not show either the specific bus or the station. Unlike B, in which operations are called, Event-B defines events that might be executed/triggered when the guard is true. For instance, in order for a bus to leave a station (*leave* event), the number of parked buses must be greater than 0. The guard of the event is represented as $parked > 0$, specifying that there is at least one bus parked. The meaning of an event is the meaning of the substitution ($parked := parked - 1$) in its body. The machine defines more events not shown in the figure.

The refinement of this machine (see left Figure 4.4) introduces more details to the system. It declares (in the context, right Figure 4.4) two sets, BUSES and STATIONS, representing the set of all possible buses and the set of all possible stations in the system. Additionally, it says that the cardinality of the set BUSES is equal to n and the cardinality of the set STATIONS is equal to m (constants defined in the abstract machine). The refinement machine defines another variable *busStation* that maps buses to stations, representing which bus is parked at which station. The variable *busStation* is defined as a partial injective function (denoted in Event-B as $\rightarrowtail\!\!\!\!\!+$), which enforces that a bus in the domain of *busStation* (buses parked) must be in one station only and that each station can hold just one bus. The refinement machine extends the abstract event *leave* by adding more details (it also extends other events from the abstract machine not shown in the figure). Specifically, in order for a bus b to leave a station (the clause any gives the machine implementer the opportunity to choose any value that satisfies the predicate in the guard), the bus b must be

```
machine ref1 refines abstract sees ctx2
 variables parked  busStation
 invariant
  inv1r1  busStation ∈ BUSES ⤔ STATIONS
  inv2r1  card(busStation) = parked
 events
  initialisation
   begin  act1  parked := 0
          actr1  busStation := ∅
   end
  leave extends leave
   any bwhere
    grdr1  b ∈ dom(busSta)
   then
    actr1  busSta := busSta\{b ↦ busSta(b)}
  end
end
```

```
context ctx2 extends ctx1

sets BUSES STATS

axioms
  ax1 finite(BUSES)
  ax2 finite(STATS)
  ax3 card(BUSES) = n
  ax3 card(STATS) = m
end
```

Figure 4.4: Part of an Event-B machine for MIO (left: refinement 1 machine. Right: context).

a bus of the system and needs to be parked at one station (**grdr1**). If the guard holds, the actions might be executed. Hence, the number of parked buses is decremented by one and the pair $\{b \mapsto busSta(b)\}$ is subtracted to the function *busStation*, indicating that bus b left the station where it was parked (**actr1**).

4.3.2 The JML-annotated Java abstract class

We used EventB2Jml tool to generate a Java abstract class of the MIO Event-B model. Figure 4.5 depicts an excerpt of the output generated by the tool. It defines Event-B carrier sets, constants, and variables with **`model`** JML clause so user can attached those variables to an actual implementation. Carrier and constants are defined with a JML **`constraint`** that prevents them to mutate their values.

As validation step, the generated JML specifications (partially depicted by Figure 4.5) was executed using the jmle tool [66, 40], validating the syntax and type correctness of the generated file. The jmle tool translates JML specifications to constraint programs, which can then be run using the Java Constraint Kit (JCK) [1]. Methods in the generated constraint programs can be called from ordinary Java code, so the programs can be used directly as (large and slow) Java implementations of the JML specifications they were generated from.

Figure 4.6 shows an implementation of the abstract Java class in Figure 4.6. Carrier sets and constants were defined as **`final`** so they cannot mutate their

```
public abstract class ref1{
   /*@ public model Integer m;
    public constraint m.equals(\old(m)); */
   /*@ public model Integer n;
    public constraint n.equals(\old(n)); */
   /*@ public model BSet<Integer> BUSES;
     public constraint BUSES.equals(\old(BUSES)); */
   /*@ public model BSet<Integer> STATS;
     public constraint STATS.equals(\old(STATS)); */

   //@ public static invariant NAT1.instance.has(n);
   //@ public static invariant NAT1.instance.has(m);
   //@ public static invariant BUSES.finite();
   //@ public static invariant STATS.finite();
   //@ public static invariant BUSES.int_size() == n;
   //@ public static invariant STATS.int_size() == m;

   //@ public model BRelation<Integer,Integer> busSta;
   //@ public model Integer parked;

   /*@ public invariant
      (new Range(0,(new BSet<Integer>(n,m)).min())).has(parked) &&
     busSta.isaFunction() && busSta.inverse().isaFunction() && busSta.domain().
          isSubset(BUSES) && busSta.range().isSubset(STATS) &&
     busSta.finite() && busSta.int_size() == parked; */

   /*@ public initially  parked == 0 && busSta.isEmpty(); */

   /*@ requires true; assignable \nothing;
     ensures \result <==> (\exists Integer b;
        (parked > 0 && busSta.domain().has(b))); */
    public abstract boolean guard_arrive();

   /*@ requires guard_leave();
     assignable parked, busSta;
     ensures (\exists Integer b;
        \old((parked > 0
        && busSta.domain().has(b)))
        &&  parked == \old(parked - 1)
        &&  busSta.equals(\old(busSta.difference((new BRelation<Integer,Integer>((
             new JMLEqualsEqualsPair<Integer,Integer>(b,busSta.apply(b)))))))));
    also
     requires !guard_leave(); assignable \nothing;
     ensures true; */
    public abstract void run_leave();
}
```

Figure 4.5: A partial JML specification of the MIO Event-B model.

```
public class ref1_impl extends ref1{

   public static final Integer mI = 3; //@ represents m = mI;
   public static final Integer nI = 3; //@ represents n = nI;
   public static final BSet<Integer> BUSESI = new BSet<Integer>(1,2,3); //@
        represents BUSES = BUSESI;
   public static final BSet<Integer> STATSI = new BSet<Integer>(1,2,3); //@
        represents STATS = STATSI;

   public BRelation<Integer,Integer> busStaI; //@ represents busSta = busStaI;
   public Integer parkedI;  //@ represents parked = parkedI;

   @Override
   public boolean guard_leave() {
      return (parkedI > 0 && busStaI.domain().has(b));
   }

   @Override
   public void run_leave() {
      parkedI = parkedI - 1;
      busStaI = busStaI.difference((
         new BRelation<Integer,Integer>((
            new JMLEqualsEqualsPair<Integer,Integer>(b,busStaI.apply(b))))));
   }
}
```

Figure 4.6: An implementation for the abstract Java class presented in Figure 4.5.

value. All variables contain the JML **`represents`** that binds the abstract variable with the actual definition.

4.4 Conclusion

In this chapter we presented a set of syntactic rules to translate Event-B models to JML specifications. We also introduced the implementation of these rules as the EventB2Jml tool which is a Rodin's plug-in. We validated EventB2Jml by applying it to a model of a transportation system (MIO) written in Event-B. Then, we manually wrote Java code from the JML specifications. Working with EventB2Jml suggests us that software developers can find the tool appealing to the development of software, specially to develop critical software. One of the advantages of EventB2Jml is that it enables users to first model the system in Event-B where the user can prove the system consistent, to then transition to JML specifications, where the user can manually write Java code. Our experience also suggests that EventB2Jml makes the use of Event-B formal method more popular since the user does not have to refine the Event-B model until an implementation, which is heavy burden, rather the user decides the level of detail in Event-B to then translate the model to JML.

As a validation step, we applied EventB2Jml to an Event-B model, this gave us the insight that EventB2Jml provides a relatively quick and easy way to generate a Java implementation from an Event-B model. However, we need to apply our tool in a wider variety of models. We found out that the process of manually generating Java code from the JML specifications generated by EventB2Jml can be optimised: the manual Java code closely follows the JML specification. We decided, instead of maintaining EventB2Jml, to upgrade the tool so it can automatically generate Java code along with the JML specification. Thus, the user do not have to spend time on manually writing the Java code. Chapter 5 discusses this upgrade.

Chapter 5

Translation of Event-B Machines to JML-annotated Java Code

This chapter. The work presented in previous chapters show how this thesis evolved in time: we started first by proposing a translation from B to JML; then we realised that Event-B is a better starting point for the modelling of critical software, so we proposed a translation from Event-B machines to JML; in the process, we proposed a translation of Event-B Proof Obligations (PO) to Dafny, so users can assist the process of discharging PO by translating them to Dafny; finally, we realised that the manual process of implementing Java code for the JML specifications generated by the EventB2Jml tool was error-prone and time consuming. Hence, this chapter presents a translation of Event-B machines to JML-annotated Java classes. It also presents the implementation of the underlying translation rules as the EventB2Java tool which is a Rodin plug-in.

Work done in this chapter is based on author's paper [91], co-authored by N. Cataño, a submitted journal paper [92], and a book chapter to appear in [36]. The rest of this chapter is organised as follows: Section 5.1 presents the translation from Event-B to JML-annotated Java programs, and Section 5.3 presents the implementation of the EventB2Java tool. Section 5.4 shows an example of applying EventB2Java to an Event-B model. Section 5.5 proposes two software development strategies using the EventB2Javatool. Finally, Section 5.6 concludes and mentions future work.

Contributions. The main contributions of this work are *i*) the definition of a full-fledged translation from Event-B to JML-annotated Java programs, and *ii*) the implementation of this translation as the EventB2Java tool. The EventB2Java Java code generator largely supports Event-B's syntax. A first key feature of this translation is that it can be applied to both abstract and refinement machines. Hence, EventB2Java tool users can generate code for a very abstract (and incomplete) Event-B model of a system, check user's intention in Java - whether the system behaves as expected, and then continue developing the Event-B model to correct any issues and add additional functionality as needed. EventB2Java can generate both sequential and multi-threaded Java implementations of Event-B models.

A second key feature of this translation is the generation of (JML) formal specifications along with the Java code. This feature enables users to write custom code that replaces the code generated by EventB2Java, and then use existing JML tools to verify that the custom code is correct.

Related work. In [77], Méry and Singh present the EB2ALL tool-set that includes the EB2C, EB2C++, EB2J, and EB2C$^\sharp$ plug-ins, each translating Event-B machines to the indicated language. Unlike EventB2Java, EB2ALL supports only a small subset of Event-B's syntax, and users are required to write a final Event-B implementation refinement in the syntax supported by the tool. In [83], Ostroumov and Tsiopoulos present the EHDL prototype tool that generates VHDL code from Event-B models. The tool supports a reduced subset of Event-B's syntax and users are required to extend the Event-B model before it can be translated. In [103], Wright defines a B2C extension to the Rodin platform that translates Event-B models to C code. The Code Generation tool [54] generates concurrent Java and Ada programs for a *tasking* extension [55] of Event-B. As part of the process of generating code with the Code Generation tool, users have to decompose the Event-B model by employing the Machine Decomposition plug-in. The decomposed models are refined and non-deterministic assignments are eliminated. Finally, users are asked to model the flow of the execution of events in the tasking extension. EventB2Java differs from all of these tools in that EventB2Java does not require user intervention before code generation, and can translate a much larger subset of Event-B syntax.

In [48], Damchoom presents a set of rules that translate Event-B to Java. However, the rules account for only a small part of Event-B's syntax and have not been implemented. Toom et. al [98] have a similar motivation; they present Gene-Auto, an automatic code generator toolset for translating from high level modelling languages like Simulink/Stateflow and Scicos to executable code for real-time embedded systems. Their approach is to work at a higher level of abstraction when verifying a solution (in the same way that we use Event-B), and then to add implementation details.

Although the modelling of timing properties is not directly supported by Event-B, a discrete clock can certainly be designed and implemented in Event-B. In [95, 96], Mohammad Reza Sarshogh and Michael Butler introduce three Event-B trigger-response patterns, namely, deadlines, delays and expires, to encode discrete timing properties in Event-B. A "deadline" means that a set of events must respond to a particular event within a bounded time. For a "delay", the set of response events must wait for a specified period after the triggering of an event. An "expiry" pattern prevents response events from triggering after the occurrence of an event. The authors translate timing properties as invariants, guards and Event-B actions. We are interested in investigating on how our code generation framework can be extended to support timing properties in Event-B, and in encoding this extension in EventB2Java once the Rodin platform fully supports the use of discrete timing events.

The Open Group has recently undertaken an effort to produce a Real-Time Java programming language called Safety-Critical Java (SCJ) [74] that augments Java with event handlers, memory areas and a Real-Time Specification for Java [101]. The design of SCJ is organised into levels so that it facilitates the certification of Safety-Critical Systems. Providing support for the encoding of real time properties in EventB2Java might require us to use SCJ rather than Java as the implementation language for Event-B. In [42], a refinement technique for developing SCJ programs based on the Circus language is proposed. Circus is based on Z, CSP, and Timed CSP so it can be used for the modelling of safety-critical systems. However, code generation is not supported by Circus.

5.1 The translation from Event-B machines to JML-annotated Java Code

We present our translation from Event-B to Java and JML using three operators (EB2Prog, EB2Java and EB2Jml), which we define via syntactic rewriting rules. The primary operator is EB2Prog, which translates Event-B to JML-annotated Java programs. It uses EB2Java to obtain the Java part of the translation and EB2Jml to obtain the JML part. For example, Event-B invariants are translated only as JML specifications, and so the definition of EB2Jml has a rule for invariants, while EB2Java does not. On the other hand, the translation of constants includes a Java part and a JML part, so the EB2Prog rule for constants refers to both EB2Java and EB2Jml rules for constants. The translation further employs operators MOD that returns the set of variables that a Java method can assign to, Pred that translates an Event-B predicate or expression, TypeOf that returns the type of a variable or constant, FreeVar that returns the set of variables that occur free in an expression, and Stat1 and Stat2 that are used in translating Event-B machine variants.

A machine is translated as a Java class. In translating an Event-B machine, EB2Prog not only considers the information provided by the machine, but also the contexts the machine sees. Figure 5.1 presents Rule M that translates a machine *M* that sees context *ctx*. The machine is translated as a Java class that includes JML class and method specifications. The translation of the machine includes the translation of the context the machine sees. Hence, the Java translation of the machine includes the translation of carrier sets, constants, axioms and theorems declared within the machine context. It also includes the translation of variables and invariants declared within the machine. Notice that we are not translating Event-B models to a Java abstract class as explained in Chapter 4, hence the translation of carrier sets, constants, and variables does

```
  EB2Prog(sets S) = S
  EB2Prog(constants c) = C
  EB2Jml(axioms X(s, c)) = X
  EB2Jml(theorems T(s, c)) = T
  EB2Java(variables v) = V
  EB2Jml(invariants I(s, c, v)) = I
  EB2Prog(variant E(s, c, v)) = Va
  EB2Prog(events e) = E
  EB2Jml(event initialisation then A(s, c, v) end) = I1
  EB2Java(event initialisation then A(s, c, v) end) = I2
--------------------------------------------------------------- (M)
EB2Prog(
  context ctx
   sets S
   constants c
   axioms X(s, c)
   theorems T(s, c)
  end

  machine M sees ctx
   variables v
   invariants I(s, c, v)
   variant E(s, c, v)
   event initialisation then A(s, c, v) end
   events e
  end) =
E
public class M {
  X T I
  S C V

  Va

  public Lock lock = new ReentrantLock(true);

  /*@ public normal_behavior
      requires true;
      assignable \everything;
      ensures I1; */
  public M(){
    I2
    // creation of Java Threads
  }
}
```

Figure 5.1: The translation of machine M, and the context C that M sees.

not use the JML keyword **`model`** since they are already Java variables. For this reason, we are not using **`represents`** as suggested in Chapter 4.

Refinement machines are translated in the same way as abstract machines since Rodin properly adds abstract machine components to the internal representation of the refining machine. Refining and extending events (defined using refines and extends, respectively) are translated in the same manner as abstract events for the same reasons.

We translate carrier sets and constants as class attributes, and restrict those attributes for verification purposes. Hence, carrier sets are translated as class attributes with the addition of a history **`constraint`** that prevents any change in their values. As we have no type information about carrier sets, they are simply translated as sets of integers.

$$\frac{\mathsf{EB2Jml}(\mathsf{sets}\ S) = \texttt{SM} \quad \mathsf{EB2Java}(\mathsf{sets}\ S) = \texttt{SA}}{\mathsf{EB2Prog}(\mathsf{sets}\ S) = \texttt{SM}\,\texttt{SA}}\ (\mathsf{Set})$$

$$\frac{}{\mathsf{EB2Jml}(\mathsf{sets}\ S) =}\ (\mathsf{Set})$$

```
//@ public static constraint
    S.equals(\old(S));
```

$$\frac{}{\mathsf{EB2Java}(\mathsf{sets}\ S) =}\ (\mathsf{Set})$$

```
public static final BSet<Integer> S =
  new Enumerated(
    Integer.MIN_VALUE,Integer.MAX_VALUE);
```

Translation of constants follows a similar pattern to the translation of carrier sets, except that in Event-B, the values of constants are constrained by axioms. The helper operator TypeOf translates the type of an Event-B variable or constant to the Java representation of that type. Function `AxiomTheoremValue<Type>` returns a value of type `Type` that satisfies the axioms defined in the contexts the machine sees[1].

$$\frac{\mathsf{EB2Jml}(\mathsf{constants}\ c) = \texttt{CM} \quad \mathsf{EB2Java}(\mathsf{constants}\ c) = \texttt{CA}}{\mathsf{EB2Prog}(\mathsf{constants}\ c) = \texttt{CM}\,\texttt{CA}}\ (\mathsf{Cons})$$

$$\frac{}{\mathsf{EB2Jml}(\mathsf{constants}\ c) =}\ (\mathsf{Cons})$$

```
//@ public static constraint
    c.equals(\old(c));
```

---

[1] Function `AxiomTheoremValue<Type>` has not yet been implemented in EventB2Java.

$$\frac{\begin{array}{l}\mathsf{TypeOf}(c) = \texttt{Type} \\ \texttt{val = AxiomTheoremValue<Type>()}\end{array}}{\begin{array}{l}\mathsf{EB2Java}(\mathsf{constants}\ c) = \\ \texttt{\textbf{public static final}}\ \texttt{Type c = val;}\end{array}}\ (\mathsf{Cons})$$

As axioms are mainly used to specify properties of constants and carrier sets, they are translated as **`static invariant`**s. A JML **`static invariant`** can only refer to static fields, and so this approach is consistent with our translation of constants and carrier sets as **`static`** fields. Translating axioms to **`static invariant`**s makes it clearer that they should not refer to machine variables, for example. In Event-B, theorems that appear in contexts should be provable from axioms, matching the semantics of the **`invariant_redundantly`** clause in JML.

$$\frac{\mathsf{Pred}(X(s,c)) = \texttt{X}}{\begin{array}{l}\mathsf{EB2Jml}(\mathsf{axioms}\ X(s,c)) = \\ \texttt{//@}\,\texttt{\textbf{public static invariant}}\ \texttt{X;}\end{array}}\ (\mathsf{Axiom})$$

$$\frac{\mathsf{Pred}(T(s,c)) = \texttt{T}}{\begin{array}{l}\mathsf{EB2Jml}(\mathsf{theorems}\ T(s,c)) = \\ \texttt{//@}\,\texttt{\textbf{public static invariant\_redundantly}}\ \texttt{T;}\end{array}}\ (\mathsf{Thm})$$

Machine variables are translated to class attributes. The JML keyword **`spec_public`** makes a **`protected`** or **`private`** attribute or method **`public`** to any JML specification.

$$\frac{\mathsf{TypeOf}(v) = \texttt{Type}}{\begin{array}{l}\mathsf{EB2Prog}(\mathsf{variables}\ v) = \\ \texttt{/*@}\,\texttt{\textbf{spec\_public}}\,\texttt{*/}\ \texttt{\textbf{private}}\ \texttt{Type v;}\end{array}}\ (\mathsf{Var})$$

In Event-B, every event must maintain the machine invariants. In JML, invariants state properties that must hold in every visible system state, specifically after the execution of the class constructor and after a method terminates. This is semantically equivalent to conjoining the invariant to the post-condition of each method and the constructor. Since the initialisation event translates to the post-condition of the class constructor (see below), and the actions of each other event translate as the post-condition of an "atomic" `run_evt` method (in Figure 5.2), Event-B invariants are naturally translated as JML **`invariants`**.

$$\frac{\mathsf{Pred}(I(s,c,v)) = \texttt{I}}{\begin{array}{l}\mathsf{EB2Jml}(\mathsf{invariants}\ I(s,c,v)) = \\ \texttt{//@}\,\texttt{\textbf{public invariant}}\ \texttt{I;}\end{array}}\ (\mathsf{Inv})$$

Machines include a specialised initialisation event that gives initial values to state variables. This event is translated by EB2Jml as the post-condition of the (only) constructor for the Java class resulting from the translation of the machine, and by EB2Java as the body of that constructor. Both translations give initial values to the translation of the machine variables.

$$\frac{\mathsf{EB2Jml}(A(s,c,v)) = \mathtt{A}}{\mathsf{EB2Jml}(\mathsf{event}\ \mathit{initialisation}\ \mathsf{then}\ A(s,c,v)\ \mathsf{end}) = \mathtt{A}}\ (\mathsf{Init})$$

$$\frac{\mathsf{EB2Java}(A(s,c,v)) = \mathtt{A}}{\mathsf{EB2Java}(\mathsf{event}\ \mathit{initialisation}\ \mathsf{then}\ A(s,c,v)\ \mathsf{end}) = \mathtt{A}}\ (\mathsf{Init})$$

Other (non-initialisation) Event-B events can be either ordinary, convergent or anticipated. Convergent events are used for modelling terminating systems. Anticipated events denote some abstract behaviour that is to be made precise in a future refinement. Convergent events must monotonically decrease the machine variant (a given natural number expression), and anticipated events cannot increase the machine variant. Events that are convergent or anticipated are only enabled if the value of the variant is non-negative. An Event-B variant expression "variant $E(s,c,v)$" is translated by EB2Prog as a method that returns the result of evaluating the translation of E.

$$\frac{\mathsf{Pred}(E(s,c,v)) = \mathtt{E}}{\mathsf{EB2Prog}(\mathsf{variant}\ E(s,c,v)) =}\ (\mathsf{Variant})$$

```
/*@ public normal_behavior
    requires true;
    assignable \nothing;
    ensures \result == E; */
public /*@ pure */ int variant() {
  return E;
}
```

Rules Status1 and Status2 below are used (by rules Any in Figure 5.2, page 72, and When in Figure 5.3, page 73) to impose the conditions associated with variants on the guards and actions of convergent and anticipated events. Translating variant expressions in this manner allows the user to verify that a customised method implementation is consistent with the meaning of the translated event – for example, since the translation of a convergent event refers to the translation of the variant in the post-condition of its JML specification, the user can employ JML machinery to verify that the customised implementation does in fact decrease the variant. The return type of method `variant()` above is **`int`**. This and the use of rule Status1 in Rule Any (and When) ensure that the variant is a natural number expression as required by Event-B[2].

---

[2] We translate variants to JML and Java since we came to realise that users might customise the Java code and after customisation users must be able to check that the customisation of a convergent event monotonically decreases the machine variant and the customisation of an anticipated event does not increase it. Not as suggested in Chapter 3.

$$\frac{}{\begin{array}{l}\mathsf{Stat1(status\ ordinary)} =\\ \quad \texttt{\textbf{true}}\end{array}}\ (\mathsf{Status1})$$

$$\frac{}{\begin{array}{l}\mathsf{Stat1(status\ convergent)} =\\ \quad \texttt{m.variant()} >= 0\end{array}}\ (\mathsf{Status1})$$

$$\frac{}{\begin{array}{l}\mathsf{Stat1(status\ anticipated)} =\\ \quad \texttt{m.variant()} >= 0\end{array}}\ (\mathsf{Status1})$$

$$\frac{}{\begin{array}{l}\mathsf{Stat2(status\ ordinary)} =\\ \quad \texttt{\textbf{true}}\end{array}}\ (\mathsf{Status2})$$

$$\frac{}{\begin{array}{l}\mathsf{Stat2(status\ convergent)} =\\ \quad \texttt{m.variant()} < \texttt{\textbf{\textbackslash old}(m.variant())}\end{array}}\ (\mathsf{Status2})$$

$$\frac{}{\begin{array}{l}\mathsf{Stat2(status\ anticipated)} =\\ \quad \texttt{m.variant()} <= \texttt{\textbf{\textbackslash old}(m.variant())}\end{array}}\ (\mathsf{Status2})$$

Standard (non-initialisation) events are translated as Java threads. In Event-B, non-mutually exclusive event guards allow the interleaving of the execution of events whereas mutually exclusive guards force events to run sequentially. We translate the latter case (see Section 5.1.2) without overriding the `run()` method, forcing the implementation to run sequentially. We translate the former case by overring the method `run()` as explained in the following. The translation of a standard event is defined by Rules Any in Figure 5.2, and when in Figure 5.3: Rule Any refers to Event-B events with local variables bounded by the Event-B clause any. Each such event is translated as a subclass of the Java `Thread` class that includes a reference to the machine class implementation. The class implementing the event contains three methods: a `guard_evt` method that tests if the guard of the event *evt* holds, a `run_evt` method that models the execution of *evt*, and a `run()` method that overrides the corresponding Java `Thread` method. Method `run_evt` is atomic – it is executed within `lock` and `unlock` instructions using a `Reentrant` lock from the Java `concurrent` Library (Section 5.3.3 explains our decision of using `Reentrant` lock rather than Java **`synchronized`** methods or an implementation of the Bakery algorithm).

Variables bounded by the any construct are translated as parameters of the `run_evt` and `guard_evt` methods (see Rule Any). The expression `GuardValue<Type>.next()` in method `run()` returns a random value of type `Type` that might satisfy the event guard. The helper operator MOD computes the set of variables assigned to in the actions of the event.

The JML specification of `run_evt` uses two specification cases. In the first case, the translation of the guard is satisfied (and the current value of the variant is non-negative for convergent and anticipated events), and the post-state of the method must satisfy the translation of the event actions and the translation of the variant restriction. In the second case, the translation of the guard is not satisfied, and the method is not allowed to modify any fields, ensuring that the post-state is the same as the pre-state. This matches the semantics of Event-B: if the guard of an event is not satisfied, the event cannot execute and hence cannot modify the system state.

An event body consists of potentially many deterministic and non-deterministic assignments. In Event-B, the symbol $:\mid$ represents non-deterministic assignment. Non-deterministic assignments generalise deterministic assignments (formed with the aid of $:=$). For example, $v := v + w$ can be expressed as $v :\mid v' = v + w$, where v' is the value of v after the assignment. The first Rule NAsg and the first Rule Asg below translate non-deterministic and deterministic assignments to JML (respectively). They are used within JML method post-conditions. The JML translation of a non-deterministic assignment $v :\mid P$ is a JML existentially quantified expression. The expression `\old(P)` ensures that `P` is evaluated in the method pre-state. This matches the Event-B semantics for assignments, in which the left-hand side is assigned the value of the right-hand side evaluated in the pre-state. The expressions `v.equals(v')` and `v.equals(\old(E))` ensure that the value v' of a variable v in the post-state is properly characterised.

The second Rule NAsg and the second Rule Asg below translate non-deterministic and deterministic assignments to Java (respectively). They are used by rules Any and When to translate the body of an event. `PredicateValue<Type>(P)` returns a value of type `Type` that satisfies predicate `P`[3].

$$\frac{\mathsf{Pred}(P(s,c,v,v')) = \texttt{P} \quad \mathsf{TypeOf}(v) = \texttt{Type}}{\begin{array}{c}\mathsf{EB2Jml}(v :\mid P) = \\ (\backslash\textbf{exists}\,\texttt{Type v'; }\backslash\textbf{old}\texttt{(P) \&\& v.equals(v'))}\end{array}}\ (\mathsf{NAsg})$$

$$\frac{\begin{array}{l}\mathsf{TypeOf}(v) = \texttt{Type} \\ \texttt{val = PredicateValue<Type>(P)}\end{array}}{\begin{array}{c}\mathsf{EB2Java}(v :\mid P) = \\ \texttt{v = val;}\end{array}}\ (\mathsf{NAsg})$$

$$\frac{\mathsf{Pred}(E(s,c,v)) = \texttt{E}}{\mathsf{EB2Jml}(v := E) = \texttt{v.equals(}\backslash\textbf{old}\texttt{(E));}}\ (\mathsf{Asg})$$

---

[3] The EventB2Java tool does not yet implement function `PredicateValue`.

$$\frac{\begin{array}{ll} \mathsf{EB2Jml}(A(s,c,v,x)) = \texttt{A} & \mathsf{EB2Java}(A(s,c,v,x)) = \texttt{B} \\ \mathsf{Pred}(G(s,c,v,x)) = \texttt{G} & \mathsf{MOD}(A(s,c,v,x)) = \texttt{D} \\ \mathsf{Stat1}(\mathit{status}\ St) = \texttt{St1} & \mathsf{TypeOf}(x) = \texttt{Type} \\ \mathsf{Stat2}(\mathit{status}\ St) = \texttt{St2} & \end{array}}{\begin{array}{c} \mathsf{EB2Prog}(\mathsf{event}\ evt \\ \mathsf{status}\ St \\ \mathsf{any}\ x\ \mathsf{where}\ G(s,c,v,x) \\ \mathsf{then}\ A(s,c,v,x)\ \mathsf{end}) = \end{array}}\ (\mathsf{Any})$$

```
public class evt extends Thread {
  private M m;

  /*@ public normal_behavior
      requires true;
      assignable \everything;
      ensures this.m == m; */
  public evt(M m){
    this.m = m;
  }

  /*@ public normal_behavior
      requires true;
      assignable \nothing;
      ensures \result <==> G && St1; */
  public /*@ pure */ boolean guard_evt(Type x) {
    return G && St1;
  }

  /*@ public normal_behavior
      requires guard_evt(x);
      assignable D; ensures A && St2;
    also
      requires !guard_evt(x);
      assignable \nothing; ensures true; */
  public void run_evt(Type x) {
    if (guard_evt(x)) { B }
  }

  public void run() {
    while(true) {
      Type x = GuardValue<Type>.next();
      m.lock.lock();
      run_evt(x);
      m.lock.unlock();
    }
  }

}
```

Figure 5.2: The translation of a standard Event-B event with local variables.

$$\frac{\begin{array}{ll}\mathsf{EB2Jml}(A(s,c,v)) = \mathtt{A} & \mathsf{EB2Java}(A(s,c,v)) = \mathtt{B}\\ \mathsf{Pred}(G(s,c,v)) = \mathtt{G} & \mathsf{MOD}(A(s,c,v)) = \mathtt{D}\\ \mathsf{Stat1}(\mathit{status}\ St) = \mathtt{St1} & \mathsf{Stat2}(\mathit{status}\ St) = \mathtt{St2}\end{array}}{\begin{array}{c}\mathsf{EB2Prog}(\mathsf{event}\ \mathit{evt}\\ \mathsf{status}\ St\\ \mathsf{when}\ G(s,c,v)\\ \mathsf{then}\ A(s,c,v)\ \mathsf{end})\ =\end{array}}\ (\mathsf{When})$$

```
public class evt extends Thread {
  private M m;

  /*@ public normal_behavior
      requires true;
      assignable \everything;
      ensures this.m == m; */
  public evt(M m){
    this.m = m;
  }

  /*@ public normal_behavior
      requires true;
      assignable \nothing;
      ensures \result <==> G && St1; */
  public /*@ pure */ boolean guard_evt() {
    return G && St1;
  }

  /*@ public normal_behavior
      requires guard_evt();
      assignable D; ensures A && St2;
    also
      requires !guard_evt();
      assignable \nothing; ensures true; */
  public void run_evt() {
    if (guard_evt()) { B }
  }

  public void run() {
    while(true) {
      m.lock.lock();
      run_evt();
      m.lock.unlock();
    }
  }

}
```

Figure 5.3: The translation of a standard event.

$$\frac{\mathsf{Pred}(E(s,c,v)) = \texttt{E}}{\begin{array}{l}\mathsf{EB2Java}(v\ := E) = \\ \quad \texttt{v = E;}\end{array}}\ (\mathsf{Asg})$$

Simultaneous assignments in the body of an event are translated individually and the results are conjoined. Assignments translate to both JML and Java. For example, a pair of simultaneous actions $x := y \mid\mid y := x$ is translated to the JML post-condition `x == \old(y) && y == \old(x)` for variables x and y of type integer.

$$\frac{\begin{array}{c}\mathsf{EB2Jml}(A1) = \texttt{A1} \\ \mathsf{EB2Jml}(A2) = \texttt{A2}\end{array}}{\mathsf{EB2Jml}(A1\ \mid\mid\ A2\ ) = \texttt{A1 \&\& A2}}\ (\mathsf{ParAsg})$$

$$\frac{\begin{array}{c}\mathsf{EB2Java}(A1) = \texttt{A1} \\ \mathsf{EB2Java}(A2) = \texttt{A2}\end{array}}{\mathsf{EB2Java}(A1\ \mid\mid\ A2\ ) = \texttt{A1 A2}}\ (\mathsf{ParAsg})$$

In Java, simultaneous actions are implemented by first calculating the value of each right hand side of the assignment into a temporary variable. The Java translation of $x := y \mid\mid y := x$ is:

```
TypeOf(x) x_temp = x;
TypeOf(y) y_temp = y;
x = y_temp;
y = x_temp;
```

Some Event-B constructs do not translate to Java or JML for various reasons. For example, consider the with construct that is used in the definition of a refinement event as a "witness" of a disappearing abstract (refined) event variable. A witness predicate specifies how the disappearing variable is implemented by the refinement event. A witness plays a similar role for an event as a "gluing invariant" does for a machine. A witness for an abstract event variable x is a predicate $P(x)$ involving x. A deterministic witness for a variable x is an equality predicate $x = E$, where E is an expression free of x. As Rodin ensures that x does not appear in the refinement event (x is replaced by E), we do not need to translate witnesses to Java or JML.

5.1.1 The Helper Operators

In the following, we present the helper operators MOD, Pred, and TypeOf used in the translation (they closely follow the definition of these operators in previous chapters). The MOD operator *collects* the variables assigned by Event-B actions[4]. The cases of MOD for assignments are shown below. For the body

---

[4] MOD's rules largely follow the syntactic rules of the Chase tool [31].

of an event, MOD is calculated by the union of the variables assigned by all contained actions.

$$\mathsf{MOD}(v\ :=E\ ) = \{v\} \quad \mathsf{MOD}(v :\mid\ P) = \{v\}$$

The Pred operator translates predicates, boolean, relational and arithmetic expressions in the natural way. To simplify the translation process, we implemented (and developed full JML specifications for) classes `BSet` and `BRelation`, representing Event-B sets and relations. In particular, these classes provided a convenient mechanism for implementing the operations on these types. Here, we present several of the rules defining Pred for applications of Event-B set and relation operations, largely by translating them to calls of the corresponding methods. In these rules, the s_i's are sets and r is a relation.

$$\frac{\mathsf{Pred}(s_1) = \texttt{s1} \quad \mathsf{Pred}(s_2) = \texttt{s2}}{\mathsf{Pred}(s_1 \subseteq s_2) = \texttt{s1.isSubset(s2)}}\ (\mathsf{Subset})$$

$$\frac{\mathsf{Pred}(x) = \texttt{x} \quad \mathsf{Pred}(s) = \texttt{s}}{\mathsf{Pred}(x : s) = \texttt{s.has(x)}}\ (\mathsf{Has})$$

$$\frac{\mathsf{Pred}(r) = \texttt{r} \quad \mathsf{Pred}(s) = \texttt{s}}{\mathsf{Pred}(r[s]) = \texttt{r.image(s)}}\ (\mathsf{Image})$$

EB2Jml translates Event-B set comprehension expressions to JML (see Rule Set-Comp) set comprehensions. Operator FreeVar returns the set of variables that occur free in an expression. The rules shows the different ways of expressing set comprehension in Event-B and the translation for each. For simplicity, we assume that E contains a single free variable x in the second rule, and that E and P do not contain a variable named e in either rule (i.e. $e \notin \mathsf{FreeVar}(E) \wedge e \notin \mathsf{FreeVar}(P)$). We do not translate set comprehensions to Java code since it is not possible in general – set comprehensions can denote infinite sets.

$$\frac{\begin{array}{ll}\mathsf{Pred}(E) = \texttt{E} & \mathsf{TypeOf}(x) = \texttt{Type}\\ \mathsf{Pred}(P) = \texttt{P} & \mathsf{TypeOf}(E) = \texttt{Type\_e}\end{array}}{\begin{array}{l}\qquad\qquad \mathsf{EB2Jml}(\{x \cdot P \mid E\}) = \\ \textbf{new}\,\texttt{BSet<Type>(}\\ \textbf{new}\,\texttt{JMLObjectSet}\ \{\texttt{Type\_e e |}\\ (\backslash\textbf{exists}\ \texttt{Type x; P; e.equals(E))}\})\end{array}}\ (\mathsf{Set\text{-}Comp})$$

$$\frac{\begin{array}{ll}\mathsf{Pred}(E) = \texttt{E} & \mathsf{TypeOf}(x) = \texttt{Type}\\ \mathsf{FreeVar}(E) = \{x\} & \mathsf{Pred}(P) = \texttt{P}\\ \mathsf{TypeOf}(E) = \texttt{Type\_e} & \end{array}}{\begin{array}{l}\qquad\qquad \mathsf{EB2Jml}(\{E \mid P\}) = \\ \textbf{new}\,\texttt{BSet<Type>(}\\ \textbf{new}\,\texttt{JMLObjectSet}\ \{\texttt{Type\_e e |}\\ (\backslash\textbf{exists}\ \texttt{Type x; P; e.equals(E))}\})\end{array}}\ (\mathsf{Set\text{-}Comp})$$

5.1.2 The Translation of Event-B to Sequential Java Programs

An event is enabled only if the event guard holds in the current state. This could be the case for several events and so the interleaving semantics of Event-B ensures that one of these events is non-deterministically selected and executed, and thus there can be just one executing at the time. On the other hand, mutually exclusive event guards force machine events to run sequentially.

The translation rules for sequential Java implementation are similar to the ones presented previously for multi-threaded Java, in which events and machines are translated as standard Java classes rather than threads[5], as shown in Rule Any_Seq.

For the execution of these sequential Java implementations we can use the framework defined in Chapter 4. The framework that enables users to experiment with Event-B models. Class `Framework` in Figure 5.5 presents a typical scheduler implementation of this behaviour, assuming that class `M` is the translation of the Event-B machine M, the sequence $x1$, ..., xn represents Event-B variables bounded by an any constructs, and the events of machine M are $evt1, evt2, \ldots, evtn$.

5.1.3 Support for Event-B Model Decomposition

When modelling systems with Event-B, one usually starts with the design of a single closed machine that includes both the system and the surrounding environment. The machine is then refined into a more concrete model of the system. Abstract machines usually include few events, variables and invariants, whereas (advanced) refinements typically contain many of them. The plethora of components in machines at later stages in the refinement chain often makes the discharge of the corresponding proof obligations in Rodin rather intricate. In certain cases an Event-B model may be regarded as being composed of two semi-independent sub-models in the sense that variables and the events affecting them in the integrated model could, in principle, be neatly split between those two sub-models. In this case, it would be very useful to provide a machine decomposition mechanism that allows one to construct two independent machines whose combined behaviour could nevertheless be provably shown to correspond to the integrated model. In [5], J.-R. Abrial and S. Hallerstede propose a technique for machine decomposition based on shared variables in which each decomposed machines simulates the behaviour of other decomposed machines through the use of *external* events. In [27] M. Butler proposes a technique for machine decomposition by shared events in which decomposed machines include copies of all of the variables that events in that machine use. The latter technique is implemented in Code Generation [54]. Both machine decomposition techniques produce independent machines that include local copies of shared variables or local events that simulate the effect of other decomposed machines acting on the

---

[5] The EventB2Java tool permits users to select between a multi-threaded or sequential Java implementation.

$$\frac{\begin{array}{ll}\mathsf{EB2Jml}(A(s,c,v,x)) = \texttt{A} & \mathsf{EB2Java}(A(s,c,v,x)) = \texttt{B}\\ \mathsf{Pred}(G(s,c,v,x)) = \texttt{G} & \mathsf{MOD}(A(s,c,v,x)) = \texttt{D}\\ \mathsf{Stat1}(\mathit{status}\ St) = \texttt{St1} & \mathsf{TypeOf}(x) = \texttt{Type}\\ \mathsf{Stat2}(\mathit{status}\ St) = \texttt{St2} & \end{array}}{\begin{array}{c}\mathsf{EB2Prog}(\mathsf{event}\ evt\\ \mathsf{status}\ St\\ \mathsf{any}\ x\ \mathsf{where}\ G(s,c,v,x)\\ \mathsf{then}\ A(s,c,v,x)\ \mathsf{end})\ =\end{array}}\ (\mathsf{Any\_Seq})$$

```
public class evt {
  private M m;

  /*@ public normal_behavior
      requires true;
      assignable \everything;
      ensures this.m == m; */
  public evt(M m){
    this.m = m;
  }

  /*@ public normal_behavior
      requires true;
      assignable \nothing;
      ensures \result <==> G && St1; */
  public /*@ pure */ boolean guard_evt(Type x) {
    return G && St1;
  }

  /*@ public normal_behavior
      requires guard_evt(x);
      assignable D; ensures A && St2;
    also
      requires !guard_evt(x);
      assignable \nothing; ensures true; */
  public void run_evt(Type x) {
    if (guard_evt(x)) { B }
  }
}
```

Figure 5.4: The translation of a standard Event-B event with local variables to a sequential Java program.

```
public class Framework {
  public static void main(String[] args) {
    M machine = new M();
    int n = /* the number of events of M */;
    java.util.Random r = new java.util.Random();
    while (true) {
      // x1 ... xn are declared and given random values
      switch (r.nextInt(n)) {
        case 0 :
          if (machine.guard_evt1(x1))
            machine.run_evt1(x1); break;
          ...
        case n - 1 :
          if (machine.guard_evtn(xn))
            machine.run_evtn(xn); break;
      }
    }
  }
}
```

Figure 5.5: A framework for executing Java classes translated from Event-B machines in a sequential fashion.

shared variables. Since the result of decomposing a machine are valid machines, these are correctly translated by our tool.

5.1.4 Support for Code Customisation

The JML specifications generated by EventB2Java enable users to replace the generated Java code with bespoke implementations. The user can then employ existing JML tools [25] to verify the customised implementation against the JML specification generated by the EventB2Java tool. For example, the code generated by EventB2Java will represent an Event-B function variable using an instance of class `BRelation` as described earlier in this section. A developer may wish to represent this variable using a Java `HashMap` instead, as this will make looking up the value of a given domain element more efficient. After generating this customised implementation, the developer can verify it against the generated JML specification, likely making use of the existing JML specification of the `HashMap` class [45].

5.2 Proof of Soundness

To gain confidence about our translation from Event-B to JML, it is necessary to prove that the proposed rules (explained in the previous section) are indeed sound. Having a proof of soundness of our translation, the user can be sure that the JML specifications generated by our rules are modelling what the user initially modelled in Event-B. Néstor et. al. proposed an initial proof of soundness of the translation [39]. The soundness proof ensures that any state transition step of the JML semantics of the translation of some Event-B construct into JML can be *simulated* by a state transition step of the Event-B semantics of that construct. The work provides the proof for invariants and the standard Event-B initialising event. It does not include full machines or Event-B contexts.

They expressed Event-B and JML constructs as types in Event-B, then implemented the translation rules explained in the previous section (denoted by operator EB2Jml) as type transducer rules. They defined a semantics of Event-B and JML types as state transducers. And finally proved that the semantics of the JML translation of Event-B constructs is simulated by the Event-B semantics of those constructs. The soundness condition is stated as a theorem and proved interactively in Rodin.

5.3 The EventB2Java Tool

The EventB2Java tool is implemented as a plug-in of Rodin [28]. Rodin is an open-source Eclipse IDE for Event-B that provides a set of tools for working with Event-B models. Rodin comes with an API that provides extra functionality on top of its core platform so as to support the implementation of applications as plug-ins. EventB2Java uses the Rodin API to collect the information of all the components of the machine to be translated. Figure 5.6 depicts a general structure of the EventB2Java tool. Rodin is composed of several plug-ins, e.g. an editor, a proof generator, provers, and model checkers and animators [72] (Figure 5.6 depicts these plug-ins in dotted squares). EventB2Java is another plug-in for Rodin. It takes an Event-B model and translates it to a JML-annotated Java program.

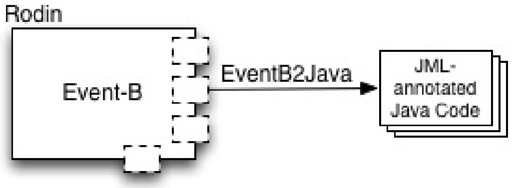


Figure 5.6: General structure of EventB2Java Rodin plug-in.

In the following we describe the structure of the EventB2Java plug-in in full detail. We first describe the structure of the Rodin platform and its main components, and then describe how the EventB2Java plug-in interfaces with these components to produce JML-annotated Java code. EventB2Java relies on a series of *recommended* interfaces [94] to interface with the Rodin components.

5.3.1 EventB2Java Rodin Plug-in Structure

Figure 5.7 shows the main components of Rodin, shown as `org.*` squares. It also shows the relation among those components and the EventB2Java

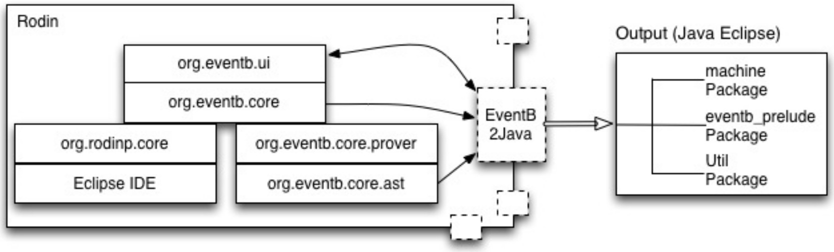


Figure 5.7: Specific structure of EventB2Java Rodin plug-in

plug-in (the solid arrows). Rodin is built on top of the Eclipse IDE. The `org.rodinp.core` component implements the core functionality of Rodin, e.g. a database for manipulating Event-B models, and for storing elements such as proof obligations and proofs. It further includes a static checker, a proof obligation generator and a prover. The `org.eventb.core.ast` component includes a library for manipulating mathematical formulas in the form of Abstract Syntax Trees. It provides an abstract class (a Visitor) for parsing the mathematical formulas. The Sequent Prover (`org.eventb.core.seqprover`) component contains a library for proving sequents. And the Event-B User Interface (`org.eventb.ui`) component contains the Graphic User Interfaces that allows users to write Event-B models and to interact with the interactive proof engine.

EventB2Java uses the Rodin `org.eventb.ui` component to manipulate context menus, e.g. to enable users to choose the type of implementation (sequential or multi-threaded) to be generated (see Figure 5.8). The relation between the `org.eventb.ui` component and EventB2Java is depicted in Figure 5.7 with a double-headed arrow: from the component to EventB2Java to capture the user's request; and from EventB2Java to the component to show the code generated.

Event-B models can include *contexts* and *machines*. Contexts can include carrier sets, constants, and axioms. Machines can include variables, invariants, and events. EventB2Java uses the Rodin `org.eventb.core` component to collect all the information of the machine and context to be translated, i.e. carrier sets, constants, axioms, variables, invariants and events. Figure 5.7 represents the relation between this component and EventB2Java with a single-headed arrow since our tool does not change the Event-B model, it just reads it.

In Event-B, models are expressed using mathematical language. The `org.eventb.core.ast` component encodes Event-B's mathematical language as nodes of an Abstract Syntax Tree (AST). This component provides various services such as parsing a formula (that is, computing its AST from a string of characters), pretty-printing a formula, constructing new formulas directly using the API library, type-checking formulas (that is, inferring the types of the expressions occurring within and decorating them with their types), testing formulas for equality, among others. EventB2Java uses the parsing service provided by this component to parse mathematical formulas to be translated to JML-annotated Java code.

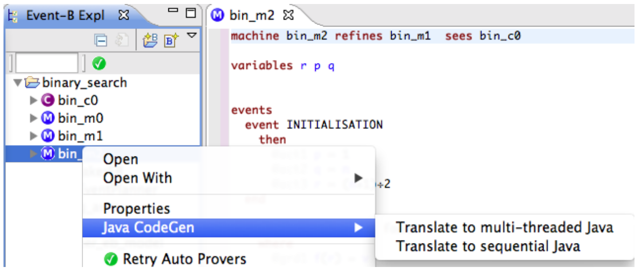
The `org.eventb.core.ast` component implements a library to traverse trees (a Visitor) and to attach information to tree nodes. Figure 5.7 uses a single-headed arrow between the `org.eventb.core.ast` component and our plug-in since the formulas are not changed. The input to `org.eventb.core.ast` is part of the information collected from the `org.eventb.core` component. EventB2Java extends the Visitor to traverse the abstract syntax trees and produce Java code and the JML specifications. Since Event-B includes mathematical notations that are not built-in to Java or JML, we implemented them as Java classes. The implementation allows EventB2Java to support Event-B's syntax (described in Section 5.3.2, and Appendix A shows for each Event-B syntax the translation to JML and Java).

After collecting the information of the Event-B contexts and machines and parsing them using the Visitor implementation, EventB2Java generates an Eclipse Java project. This project contains various packages: The `machine` package contains the translation of the machines and contexts. This package includes a main Java class with information about carrier sets, constants, and variables from the Event-B model. It also contains JML specifications generated from axioms and invariants in Event-B. This package also contains the translation of each event and a test file to run the generated Java implementation.

The Eclipse project generated by EventB2Java further includes an `eventb_prelude` package that contains the Java classes necessary to support all the Event-B syntax as explained in the next section. Finally, the `Util` package in the Eclipse project generated by EventB2Java includes utility methods. For instance, it includes an implementation of a `SomeVal` method that returns a random value contained within a set. It also includes the implementation of a `SomeSet` method that returns a random subset of a set.

EventB2Java is available at `http://poporo.uma.pt/EventB2Java`. This web site includes detailed instructions on how to install and use the tool. The EventB2Java Eclipse plug-in's update site is `http://poporo.uma.pt/Projects/EventB2JavaUpdate`, and EventB2Java has been tested on Rodin version 2.8.

EventB2Java Tool Usage: In a typical interaction with EventB2Java, a user right-clicks an Event-B machine in the Explorer panel of Rodin and selects "Translate to multi-threaded Java" or "Translate to sequential Java" (as shown in Figure 5.8). EventB2Java generates an Eclipse project that includes the JML-annotated Java implementation of the machine and the libraries needed to execute the Java code. This Eclipse project is available in the "Resource" perspective of Rodin. The Eclipse project includes a folder that contains the generated code, and an "eventb_prelude" sub-folder that contains the libraries implementing sets and relations in Java.



```
machine bin_m2 refines bin_m1  sees bin_c0

variables r p q


events
  event INITIALISATION
    then
```

Figure 5.8: EventB2Java: Contextual menu in Rodin

5.3.2 Java Implementation of Event-B Mathematical Notations in EventB2Java

The Event-B modelling language is composed of five mathematical languages (see Chapter 9 of [4]), namely, *a*) a Propositional Language, *b*) a Predicate Language, *c*) an Equality Language, *d*) a Set-Theoretic Language, and *e*) Boolean and Arithmetic Languages. Each language defines a series of constructs to model systems. To provide support for the translation from Event-B, we have implemented a series of JML-annotated Java classes; other Event-B constructs are supported natively in Java. These classes are: `BOOL`, `INT`, `NAT`, `NAT1`, `Enumerated`, `Pair`, `BSet`, `BRelation`, and `ID` (implementing, respectively, booleans, integers, natural numbers with and without 0, the enumerated type, pairs of elements, sets, relations, and the identity relation). `BSet` is implemented as a subclass of the standard Java class `TreeSet`, and `BRelation` as a set of pairs.

We had previously implemented versions of these classes, for the work described in [41] (used also in the work described in Chapter 4). Particular kinds of Event-B relations (total relations, functions, etc.) are translated as `BRelations` with appropriate restrictions added to the invariant For example, Pred($r \in D \rightarrowtail R$) for sets D and R equals: `r.isFunction() && r.inverse().isFunction() && r.domain().equals(D) && r.range().isSubset(R)`, which is added to the invariant. We further define classes `Enumerated`, `ID`, `INT`, `NAT`, `NAT1`, `Pair` and `BOOL`. For example, Pred($i \in \mathbb{N}$) = `NAT.instance.has(`i`)`, which restricts i to be non-negative. The TypeOf operator translates the type of Event-B variables and constants to the corresponding Java type. All integral types are translated as type `Integer`, all relations and functions are translated as type `BRelation`, and all other sets are translated as type `BSet`.

Some of the constructs of the Propositional Language are supported natively in Java. Negation ($\neg$) translates as `!`, conjunction ($\wedge$) as `&&`, and disjunction ($\vee$) as `||`. Other constructs such as $\Rightarrow$ and $\Leftrightarrow$ are implemented as methods of the class `BOOL`. The Predicate Language introduces constructs for universal and existential quantification. Universally and existentially quantified predicates $\forall\, x \cdot (P)$ and $\exists\, x \cdot (P)$ are translated as the JML universally and existentially quantified expressions `(\`**`forall`** TypeOf(x) `x; P)` and `(\`**`exists`** TypeOf(x) `x; P)` respectively, where P is the JML translation of P[6]. The Predicate Language also includes a construct $e \mapsto f$ that maps an expression e of type E to an expression f of type F. EventB2Java translates this construct as an instance of `Pair<E,F>`.

The Event-B Equality Language introduces equality predicates $E = F$ for expressions E and F, translated as `E.equals(F)`, if `E` and `F` are object references, or `E == F`, if they are of a primitive type. The Set-Theoretic Language introduces sets and relations in Event-B. Set operations include membership ($\in$), cartesian product ($\times$), power set ($\mathbb{P}$), inclusion ($\subseteq$), union ($\cup$), intersection ($\cap$), and difference ($\setminus$). These operations are all implemented as methods of the class `BSet`. Operations on relations in Event-B include domain restriction ($\lhd$), range restriction ($\rhd$), etc. All these operations are implemented as methods of the class `BRelation`. Relations also include notations for surjective relations $\mathrel{\leftarrow\!\!\!\twoheadrightarrow}$, total surjective relations $\mathrel{\twoheadleftarrow\!\!\!\twoheadrightarrow}$, functions, etc. EventB2Java translates all these as instances of `BRelation` with JML **`invariants`** that constrain the domain and the range of the relation, e.g. a total function is a relation in which each element in the domain is mapped to a single element in the range.

The Boolean and Arithmetic Languages define the set BOOL, containing elements TRUE and FALSE, $\mathbb{Z}$, containing the integer numbers, $\mathbb{N}$, containing the natural numbers (0 inclusive), and $\mathbb{N}_1$, containing the natural numbers (0 exclusive). EventB2Java includes implementations of these constructs in Java, namely, classes `BOOL`, `INT`, `NAT`, and `NAT1`. The Arithmetic Language defines constructs over numbers. Operators such as $\leq$, $\geq$, etc. are directly mapped into Java operators <=, >=, etc. The construct $a \mathinner{..} b$, which defines an interval between a and b, is implemented as an appropriate instance of the class `Enumerated`.

5.3.3 Decision on using `Reentrant` lock/unlock methods

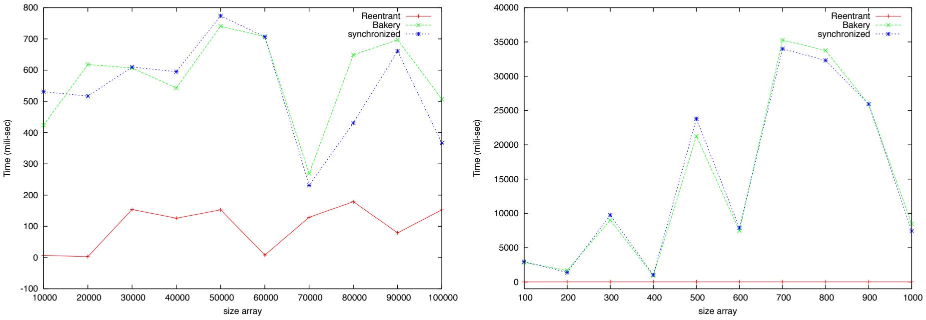
There are several ways to implement the problem of the Critical Section in Computer Science. We compared the execution times and CPU usage for three methods in order to decide which method the EventB2Java tool should use. We compared an implementation of the Bakery algorithm [69], the **`synchronized`** native Java method, and methods `lock/unlock` from the `concurrent` Java library. All experiments are available at `http://poporo.uma.pt/EventB2Java/exps.zip`

We first compared the execution times for four multithreaded Java code using the methods for implementing the critical section explained above. We used a multithreaded implementation of a Binary and Linear search in an array, the Minimum element of an array, and the Sorting algorithm of an array. We ran the implementation varying the size of the arrays. Figure 5.9 depicts the execution times taken for the four algorithms.

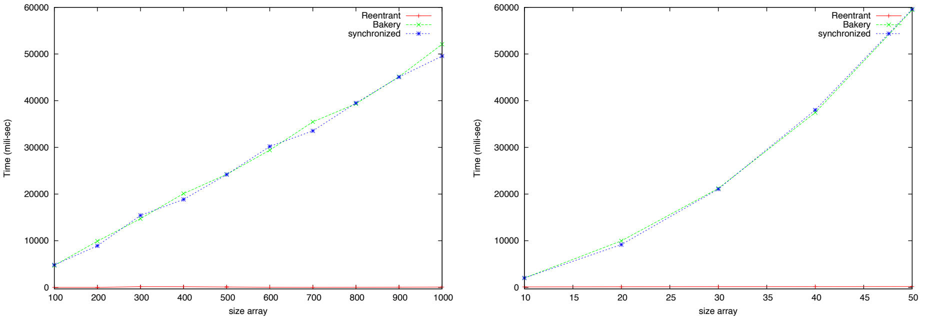
We also compared the CPU usage for them. For this experiment we used the

---

[6] EventB2Java does not generate Java code for quantified predicates.



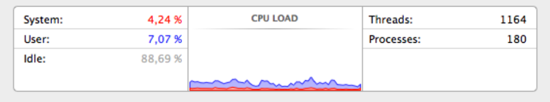
(a) Binary Search (b) Linear Search



(c) Minimum Element (d) Sorting

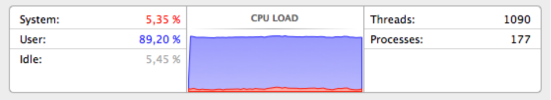
Figure 5.9: Exp 1: Execution times 'bakery' vs '**`synchronized`**' vs '`lock/unlock`'

2 multithreaded implementations that run forever. The code was ran for 5 minutes, we took the CPU usage every minute for both methods. Figure 5.10 shows the CPU usage for both method during the 5 minutes. It can be seen that the 'Bakery' method outperforms the '**`synchronized`**' method. For the 'Bakery' algorithm, in average the CPU usage was 8% whilst for the '**`synchronized`**' method was 89%.



System: 4,24 %
User: 7,07 %
Idle: 88,69 %
Threads: 1164
Processes: 180

(a) Bakery



System: 5,35 %
User: 89,20 %
Idle: 5,45 %
Threads: 1090
Processes: 177

(b) Synchronized

Figure 5.10: Exp 2: CPU usage 'bakery' vs '**`synchronized`**'

Figure 5.9 depicts that the '`lock/unlock`' method outperforms both the 'Bakery' algorithm and '**`synchronized`**' method. This is due both approaches guarantee fairness by putting each `Thread` to sleep for a random number (we used a random number from 1 to 100), while '`lock/unlock`' method guarantees fairness by passing a boolean parameter to the constructor of the class lock[7]

Our decision on using '`lock/unlock`' methods from the concurrent Java library in the EventB2Java was based on these two experiments: '`lock/unlock`' makes better use of the CPU and outperforms the time execution to both "**`synchronized`**' method and 'Bakery' algorithm. Another reason is the library guarantees fairness by itself.

5.4 Using the EventB2Java tool

We have validated our tool by applying it to an ample set of Event-B models. They are described in Section 7.3. We describe in next subsections an Event-B model modelled by J.-R Abrial in [3] and show the JML-annotated Java code generated by EventB2Java.

5.4.1 An Example in Event-B

The Binary Search algorithm finds the index of an element within a sorted array. It works by choosing a pivot index in the domain of the array and comparing the value at the index with the element one is searching for; if the value at the

---

[7] from the documentation of the class: "The constructor for this class accepts an optional fairness parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order."

```
machine $bin\_m2$ sees $bin\_c0$
 variables $r\ p\ q$
 invariant
  inv1 $p \in 1\,..\,n$
  inv2 $q \in 1\,..\,n$
  inv3 $r \in p\,..\,q$
  inv4 $v \in f[p\,..\,q]$
 events
  initialisation
   then
    act1 $p := 1$
    act2 $q := n$
    act3 $r := (n+1)$ div 2
   end
  inc
   when
    grd1 $f(r) < v$
   then
    act1 $r := (r+1+q)$ div 2
    act2 $p := r+1$
  end
  dec
   when
    grd1 $f(r) > v$
   then
    act1 $r := (p+r-1)$ div 2
    act2 $q := r-1$
  end
  found grd1 $f(r) = v$
  end
end
```

```
context $bin\_c0$
 constants $n\ f\ v$
 axioms
  axm1 $f \in 1\,..\,n \rightarrow N$
  axm2 $\forall\, i, j .\, i \in 1\,..\,n$
      $\wedge j \in 1\,..\,n \wedge$
      $i \leq j$
      $\Rightarrow$
      $f(i) \leq f(j)$
  axm3 $v \in ran(f)$
 theorem
  thm1 $n \geq 1$
end
```

Figure 5.11: An extract of the Binary Search algorithm in Event-B

index is greater than the element, then the algorithm recursively searches for the element in the sub-array to the left of the pivot, if the value is lesser than the element then the algorithm searches for the element in the sub-array to the right of the pivot; otherwise it returns the index of the element. In [3], J.-R. Abrial presents the full model of the algorithm in Event-B. Context bin_c0 in Figure 5.11 declares constants f, n, and v to be the array, its size, and the value the algorithm searches for, respectively. The correct values of these constants are axiomatised within the context representing the preconditions of the Binary Search algorithm. Axiom **axm1** declares f to be a total function. Axiom **axm2** requires f to be a sorted array, and axiom **axm2** requires the value v to exist within the array f. Theorem **thm1** can be deduced from axiom **axm1**.

The current left and right indexes of the array are given by variables p

and q, which are given initial values 1 and n, respectively. The algorithm searches for an index $p \leq r \leq q$ in the domain of f such that $f(r) = v$. These conditions are modelled as machine invariants in Event-B (see left Figure 5.11). The *initialisation* event picks up the middle value for the pivot index r. The machine *bin_m*2 declares three standard events *inc*, *dec*, and *found*. Event *inc* models the case when the value is to the right of the pivot, *dec* the case when the value is to the left of the pivot, and *found* when the value is at the pivot so no further actions are made.

5.4.2 The Generated JML-annotated Java code

Figures 5.12, and 5.13 show the JML-annotated Java code generated by EventB2Java for the Event-B model depicted in Figure 5.11. Figure 5.12 shows the translation for the machine, and Figure 5.13 depicts the event *inc* (events *dec* and *found* are not shown). EventB2Java translates Event-B constants such as f, n and v, directly into Java as **`static final`** variables. However, the tool does not generate initial values for these variables. The initial values depend on the constraints imposed by axioms and theorems. For instance, f must be a sorted array, as described by axiom **axm2**, which contains a value v, as described by axiom **axm3**. Nonetheless, EventB2Java generates JML specifications for these axioms which one can use to verify whether the initial values one conjectured for these constants are correct or not. EventB2Java translates variables as **`private`** class fields with the respective **`spec_public`** JML clause so variables can be used for verification. It also defines the corresponding getter and mutator methods (not shown in the figure). EventB2Java translates Event-B invariants as JML **`invariants`**. Finally, defines the initial values of variables according to the *initialisation* Event-B event, and creates the corresponding threads (i.e. each Event-B event is translated as a Java class that extends `Thread`).

Figure 5.13 shows the JML-annotated Java code that EventB2Java generates for event *inc*. It declares a variable `machine` which is a reference to the main Java class that contains the definition of carrier sets, constants, and variables. It defines 3 methods as explained in previous sections: the `guard_inc` method that returns **`true`** if the evaluation of applying the variable `r` to the relation `f` is less than the value `v` (the value that the algorithm is looking for). That corresponds to guard of the event *inc* in the Event-B model; the `run_inc` method that performs the actions of the event updating the variable `r` to `(r+1+q) / 2`, and variable `p` to `r + 1`. Notice that it is important to get the value of variables before the assignment (e.g. `r_tmp`), if that was not the case the variable `p` would have been assigned to a wrong value; the `run` method overriding a method from `Thread`. It implements a critical section for the execution of the method `run_inc` simulating the behaviour of executing events in Event-B.

```
public class bin_m2{
   ...
   public Lock lock = new ReentrantLock(true);

   /******Constant definitions******/
   //@ public static constraint f.equals(\old(f));
   public static final BRelation<Integer,Integer> f = Test_bin_m2.random_f;

   //@ public static constraint n.equals(\old(n));
   public static final Integer n = Test_bin_m2.random_n;

   //@ public static constraint v.equals(\old(v));
   public static final Integer v = Test_bin_m2.random_v;

   /******Axiom definitions******/
   /*@ public static invariant  f.domain().equals(new Enumerated(new Integer(1),n)
        ) && f.range().isSubset(NAT.instance) && f.isFunction() && BRelation.
        cross(new Enumerated(new Integer(1),n),NAT.instance).has(f); */
   /*@ public static invariant
      (\forall Integer i;  (\forall Integer j; ((
         new Enumerated(new Integer(1),n).has(i) &&
         new Enumerated(new Integer(1),n).has(j) &&
         i.compareTo(j) <= 0) ==> (f.apply(i).compareTo(f.apply(j)) <= 0)))); */
   /*@ public static invariant f.range().has(v); */
   /*@ public static invariant_redundantly (n).compareTo(new Integer(1)) >= 0; */

   /******Variable definitions******/
   /*@ spec_public */ private Integer p;
   /*@ spec_public */ private Integer q;
   /*@ spec_public */ private Integer r;

   /******Invariant definition******/
   /*@ public invariant
        NAT.instance.has(r) &&
        new Enumerated(new Integer(1),n).has(p) &&
        new Enumerated(new Integer(1),n).has(q) &&
        new Enumerated(p,q).has(r) &&
        f.image(new Enumerated(p,q)).has(v); */

   /*@ public normal_behavior
      requires true;
      assignable \everything;
      ensures p.equals(1) &&
        q.equals(n) &&
        r.equals(new Integer(new Integer(n + 1) / 2));*/
   public bin_m2(){
      p = 1;
      q = n;
      r = new Integer(new Integer(n + 1) / 2);

      // Threads initialisation
   }
}
```

Figure 5.12: JML-annotated Java code generated by EventB2Java from the bin_m2 depicted in Figure 5.11.

```
public class inc extends Thread{
   /*@ spec_public */ private bin_m2 machine;

   /*@ public normal_behavior
      requires true;
      assignable \everything;
      ensures this.machine == m; */
   public inc(bin_m2 m) {
      this.machine = m;
   }

   /*@ public normal_behavior
      requires true;
      assignable \nothing;
      ensures \result <==>
                machine.f.apply(machine.get_r()).compareTo(machine.v) < 0; */
   public /*@ pure */ boolean guard_inc() {
      return machine.f.apply(machine.get_r()).compareTo(machine.v) < 0;
   }

   /*@ public normal_behavior
      requires guard_inc();
      assignable machine.p, machine.r;
      ensures guard_inc() &&
            machine.get_r(),equals(\old(new Integer(new Integer(machine.get_r() +
                 1 + machine.get_q()) / 2))) &&
            machine.get_p().equals(\old(new Integer(machine.get_r() + 1)));
    also
      requires !guard_inc();
      assignable \nothing;
      ensures true; */
   public void run_inc(){
      if(guard_inc()) {
         Integer r_tmp = machine.get_r();
         Integer p_tmp = machine.get_p();

         machine.set_r(new Integer(new Integer(r_tmp + 1 + machine.get_q()) / 2));
         machine.set_p(new Integer(r_tmp + 1));
      }
   }

   public void run() {
      while(true) {
         machine.lock.lock(); // start of critical section
         run_inc();
         machine.lock.unlock(); // end of critical section
      }
   }
}
```

Figure 5.13: Binary Search: code generated for the *inc* event

```
public class bin_m2{
   ...
   public inc evt_dec = new dec(this);
   public dec evt_inc = new inc(this);
   public found evt_found = new found(this);

   ...

   public bin_m2(){
      p = 1;
      q = n;
      r = new Integer(new Integer(n + 1) / 2);
   }
}
```

Figure 5.14: Excerpt of the sequential JML-annotated Java code generated by EventB2Java from the *bin_m2* depicted in Figure 5.11.

```
public class inc extends Thread{
   /*@ spec_public */ private bin_m2 machine;

   ...
   public inc(bin_m2 m) {
      ...
   }

   ...
   public /*@ pure */ boolean guard_inc() {
      ...
   }

   ...
   public void run_inc(){
      ...
   }
}
```

Figure 5.15: Excerpt of the sequential Java code generated for the *inc* event

Notice that the execution of the Event-B model describe in Figure 5.11 is sequentially: the guards of the events are mutually exclusive. In this case, it is not necessary to translate the model to a multithreaded Java version. The EventB2Java tool allows user to generate sequential Java code for those models that their execution is sequential. Figures 5.14 and 5.15 present the sequential JML-annotated Java code generated by EventB2Java. Notice that the translation of the machine define neither threads nor the reentrant lock, and it declares the classes corresponding to the translation of the events as class fields. Notice also that the translated Java class of the event *inc* neither extends `Thread` nor overrides the `run` method.

EventB2Java generates an additional Java class defining a framework (as described in Section 5.3.2) that executes the logic of the system, Figure 5.16

```
public class Framework{
   public static void main(String[] args){
   bin_m2 machine = new bin_m2();
   int n = 3; //the number of events in the machine
   while (true){
      switch (rnd.nextInt(n)){
         case 0: if (machine.evt_found.guard_found())
               machine.evt_found.run_found(); break;
         case 1: if (machine.evt_inc.guard_inc())
               machine.evt_inc.run_inc(); break;
         case 2: if (machine.evt_dec.guard_dec())
               machine.evt_dec.run_dec(); break;
         }
      }
   }
}
```

Figure 5.16: Binary Search: Sequential Java code generated for the *inc* event

depicts it. The user can customise the proposed Framework by changing the condition of the **`while`** for `machine.evt_found.guard_found()`.

5.5 Software Development with EventB2Java

We have validated the EventB2Java tool by generating JML-annotated Java code for several Event-B models and by comparing our tool with different Java code generators for Event-B (as shown in previous section). We have validated EventB2Java also by applying it in two case studies presented in next chapter (see Chapter 7). The case studies follow two strategies for software development describe in the following. The first one, (described in Section 5.5.1), shows how EventB2Java can be used as part of a software development strategy to generate the core functionality (the Model) of an Android application that is organised following the MVC (Model-View-Controller) design pattern [60]. The second one (described in Section 5.5.2) shows how EventB2Java and Java Unit (JUnit) testing can be used to refine (improve) an Event-B model system to conform to an existing System Test Specification (STS) document.

5.5.1 Strategy on Software Development using MVC design pattern

Typical software applications include an interface (the View) that interacts with the user, a functional core (the Model) that implements the basic functionality of the application, and a linking part (the Controller) that *disguises* all requests made by the user so that they can be understood by the Model. The Model implements methods to edit data and to access the internal state of the application. It might also include a registry of dependent Views to notify when data changes occur during the *rendering* of the interface. The Controller implements wrapping code that transforms mouse input and keyboard shortcuts to commands in the Model.

EventB2Java can be used to develop a system that follows the MVC design pattern. The View-Controller components are developed using *Usability Engineering* techniques as advocated by J. Nielsen in [82]; the core functionality is modelled in Event-B and the EventB2Java tool is used to generate the Model in Java.

The strategy comprises the following steps:

1. a system is modelled in Event-B and it is refined to the desired level of abstraction via a hierarchy of machine refinements.
2. all proof obligations of the above Event-B model are discharged in Rodin, so one can be sure about the soundness of the system modelled.
3. the Event-B model is automatically translated to Java using the EventB-2Java tool.
4. the View part of the system is developed using *Usability Engineering* techniques. EventB2Java generates getter and setter methods for machine variables in the Java code generated enable communication between the Model and the View.

This strategy has been successfully applied for the development of a Social-Event Planner explained in Section 7.1.

5.5.2 Strategy on Software Testing

The use of a formal specification language to model software requirements eliminates ambiguity and reduces the chances that errors can be introduced during software development. Naturally, it still remains the issue of coming up with a formal specification that matches customer expectations, and an implementation that matches the formal specification.

EventB2Java can be used as part of a formal methods strategy for testing the behaviour of a reactive system modelled in Event-B.

The strategy comprises the following steps:

1. one models the reactive system in Event-B following an existing System Requirements Specification (SRS) document.
2. all proof obligations of the above Event-B model are discharged in Rodin, so one can be sure about the soundness of the system modelled.
3. the Event-B model is translated to Java code using EventB2Java.
4. based on an existing System Test Specification (STS) document, one manually writes Java Unit (JUnit) [73] tests that exercise the functionality of the generated Java code.

If a JUnit test fails, the Event-B model is inspected and evolved to conform to the STS document, and EventB2Java is used again to generate Java code. This testing process is repeated until all of the JUnit tests pass. The soundness of the Event-B model can then be verified by discharging proof obligations using standard Event-B tooling. After making any changes to the model needed for verification, it is translated to Java using EventB2Java, and the test suite run once more to check that the behaviour is still as expected. Note that these steps are complementary in that they address very different questions about the model – testing with EventB2Java checks whether the model behaves as the user expects, while verification checks whether the model is sound. In particular, a sound model might not have the behaviour that the developer actually intended. This testing strategy can further be combined with the use of the ProB model checker [72]. The difference is that ProB works directly on the Event-B model whilst our strategy incorporates the use of a STS document with software testing conducted in Java. That is, the strategy uncovers inconsistencies in the Event-B model vis a vis the STS (detecting whether the Event-B model captures user's intentions), while ProB determines if any valuation (a program) exists that implements the Event-B model.

This strategy has been successfully applied for the testing of the Tokeneer reactive system explained in Section 7.2.

5.6 Conclusion

In this chapter we presented a series of rules to generate JML-annotated Java code from Event-B models. We also introduced the implementation of these rules as the EventB2Java tool which is a Rodin plug-in. EventB2Java enables users to generate an actual implementation of Event-B models in Java. The Java code contains JML specifications that enable users to customise the Java code to further check if the customised code does not invalidate the initial model in Event-B. EventB2Java generates both sequential and multithreaded implementations of the models. We have validated the tool by applying it to an ample set of Event-B models to generate Java implementations. This chapter shows a small example, Chapter 7 shows the use of EventB2Java in several Event-B models.

The process followed to generate JML-annotated Java implementations from Event-B models made easier the definition of the translation rules and the implementation of the tool: we started by defining a translation from B machines to JML specification (see Chapter 3), then defining a translation from Event-B machines to JML specifications (see Chapter 4), the definition of the rules presented in this chapter and the implementation of the EventB2Java tool were relatively easy since we already had the expertise on the previous translations.

Software developers can benefit of EventB2Java since they can start the software development in Event-B, where the system is modelled and proven consistent. Once the model is correct and has the necessary detail, developers can translate it to Java using EventB2Java. Developers can customise the Java code being sure that the initial model is not invalidated since the JML specifications help them to check if the customised Java code is correct. While it is true that using Rodin users can obtain a final implementation of the models by refining the model until a very close machine implementation, it is also true that the process of refining involves a lot of mathematical expertise since every refinement machine needs to be proven consistent with the previous one. Event-B2Java makes this process easy since users can decide the level of abstraction in the Event-B model and use the tool to generate Java code.

Chapter 6

Translating Event-B Machines Proof Obligations to Dafny

This chapter. One uses Event-B for the formal modelling of critical software. In modelling in Event-B, one needs to prove the system consistent by discharging proof obligations (POs). Rodin, the Eclipse IDE for working with Event-B, automatically generates the POs. There exist different ways to discharge POs: Rodin comes with a prover (New PP) that can automatically discharge proofs; Rodin also comes with third-party provers (they need to be installed as plug-ins) to automatically discharge POs; and the user can attempt to discharge proof obligations interactively using the interactive proving that Rodin provides. The work described in this chapter seeks to help users to discharge Event-B POs by translating them to the input language of Dafny. This chapter presents a translation of Event-B proof obligations to the input language of Dafny and the implementation of the translation as the EventB2Dafny Rodin plug-in. We do not use the programming language of Dafny but the automatic verifier associated to it to discharge the generated PO. EventB2Dafny supports the full Event-B syntax. The work presented in this chapter is published in [35]. I participated in the definition of the translation rules from Event-B Proof Obligations to Dafny and its implementation. The rest of this chapter is organised as follows. The description of the type of proof obligations generated by the Rodin platform are shown in Section 6.1. Then, the translation of Rodin proof-obligations to Dafny programs is explained in Section 6.2. Section 6.3 shows the implementation of the EventB2Dafny tool. Finally, Section 6.4 gives conclusions.

Contributions. *i*) The definition of a translation from Event-B proof obligations to Dafny through a collection of rules, one for each component of an Event-B proof obligation, *ii*) the implementation of this translation as the EventB2Dafny tool. The translation allows Refinement Calculus based approaches to have access to an ample set of formal techniques. *iii*) The implementation of a prelude in Dafny language that implements sets (e.g. sets, relations, functions) as datatypes and operations over these structures as functions.

Related work. In [51], David Déharbe presents an approach to translate Rodin platform proof-obligations to the input language of the SMT-solvers. The approach handles proof obligations that include boolean expressions, integer arithmetic expressions, and basic sets and relations. The EventB2Dafny plug-in works in a similar direction. Yet, by generating bespoke Dafny/Boogie proof obligations, we can improve the performance of the Z3 SMT solver [49] on which Boogie works.

In [76], D. Mentré, C. Marché, J. Filliâtre and M. Asuka present the bpo2why tool that translates proof-obligations generated by the Atelier B suite into Why [58] programs. Therefore, proof-obligations can be discharged using the Krakatoa tool [59] or other automatic provers like Z3 [49]. The EventB2Dafny plug-in works in a similar direction as the bpo2why tool, but our target language is Dafny rather than Why.

6.1 Rodin Proof Obligations

The Rodin proof-obligation generator automatically generates proof obligations (POs) based on both the underlying machines and contexts as explained in Section 2.3.1. This section presents the type of POs generated by Rodin in a detailed way and explains some of them through an example presented by J.-R. Abrial in [4] (Chapter 5). We present proof obligations (POs) as Sequents: given the set of hypotheses H and the goal G, a Sequent is represented by

$$\begin{array}{l} H \\ \vdash \\ G \end{array}$$

A sequent reads as follows: under the hypotheses H, prove the goal G.

Figures 6.1 and 6.2 (taken from the example presented in [4] - Chapter 5) depict an Event-B example (both machines see the context shown in Figure 6.1 right). The example presents an Event-B model for searching an element in a sequence of integers. The model finds an index i of an element v in a sequence f. Context $ctx0$ in Figure 6.1 defines a constant n to be a natural number. It represents the size of the sequence f. Sequence f is defined as a total function (axiom **ax2**) that maps natural numbers (from 1 to n) to elements of set D. Constant v is defined as a natural number and axiom **ax3** states that v is present in the range of function f. The most abstract machine (Figure 6.1) models the search process in one step. The *search* event defines a local variable k that takes a value from 1 to the number of elements in the sequence. Additionally, the evaluation of the function f in k is equal to value v, so the variable i takes that value (notice that the value v is always presented in the sequence f, axiom **ax3** in the context $ctx0$ states that).

```
machine $m0\_a$ sees $ctx0$
 variables
  $i$
 invariants
  inv1 $i \in 1 \,..\, n$
 events
  $initialisation$
   then
    act1 $i := 1$
   end
  $search$
   any
    $k$ where
    grd1 $k \in 1 \,..\, n$
    grd2 $f(k) = v$
   then
    act1 $i := k$
  end
end
```

```
context $ctx0$

sets D
constants $n\ f\ v$
axioms
  ax1 $n \in \mathbb{N}$
  ax2 $f \in 1 \,..\, n \rightarrow \mathsf{D}$
  ax3 $v \in ran(f)$
theorem
  thm1 $n \in \mathbb{N}1$
end
```

Figure 6.1: An abstract and context machine in Event-B

The first refinement of the model (see Figure 6.2), introduces the search strategy. The strategy introduces variable j that starts at value 0 (see event *initialisation*) and it is then incremented by 1 (see event *progress*). According to theorem **thm1_r1**, the value v is in the evaluation from $j+1$ to n, of the sequence f. Once variable j takes the value in which the evaluation of $f(j+1)$ is equal to v (guard **grd1_r1**), the *search* event might be triggered and i will take the value $j+1$.

The set of proof obligations generated by Rodin to prove the consistency of a system are:

The Invariant Preservation Proof Obligation: it states that events must conform to machine invariants. Rodin generates a "*evt*/**inv**/INV" proof obligation that states that the event *evt* conforms the invariant **inv**. Let *evt* be as follows

```
$evt$
  any $x$
  where
    $G(s, c, v, x)$
  then
    $v :| \ BA(s, c, v, x, v')$
end
```

The sequent takes as hypotheses the axioms and theorems, the invariant,

```
machine $m1\_a$ refines $m0\_a$ sees $ctx0$
 variables $i\ j$
 invariants
  inv1_r1 $j \in 0\,..\,n$
  inv2_r1 $v \notin f[i\,..\,j]$
  thm1_r1 $v \in f[j+1\,..\,n]$
 variant $n - j$
 events
  initialisation extends initialisation
   then
    act1_r1 $j := 0$
   end
  search refines search
   when
    grd1_r1 $f(j+1) = v$
   with k: $j + 1 = k$
   then
    act1_r1 $i := j + 1$
  end
  progress
   status convergent
   when
    grd1_r1 $f(j+1) \neq v$
   then
    act1_r1 $j := j + 1$
  end
end
```

Figure 6.2: Refinement machine in Event-B

the guard of the event, and the before-after predicate, and one needs to prove that the modified invariant holds

Axioms and theorems	$A(s, c)$
Invariants	$I(s, c, v)$
Guards of the event	$G(s, c, v, x)$
Before – after predicate of the event	$BA(s, c, v, x, v')$
$\vdash$	$\vdash$
Modified specific invariant	$inv(s, c, v')$

For the example presented in Figure 6.1 Rodin generates a *search*/**inv1**/INV proof obligation that states that the new value of i (denoted i') must respect the machine invariant

$$
\begin{array}{ll}
\quad n \in \mathbb{N} & f \in 1\,..\,n \rightarrow \mathsf{D} \\
\quad v \in ran(f) & n \in \mathbb{N}1 \\
\quad i \in 1\,..\,n & k \in 1\,..\,n \\
\quad f(k) = v & i' = k \\
\vdash & \\
\quad i' \in 1\,..\,n &
\end{array}
$$

The Guard strengthening Proof Obligation: it states that the guards of the refinement machine must be stronger than the abstract event's guards. Rodin generates a "*evt*/**grd**/GRD" proof obligation that states that the guards of the refinement event *evt* are stronger that the guard **grd** of event *evt*. Let event *evt*0 and its refinement *evt* be as follows

```
$evt0$
  any $x$
  where
   grd $g(s, c, v, x)$
   ...
  then
   ...
end
```

```
$evt$ refines $evt0$
  any $y$
  where
    $H(y, s, c, w)$
  with
    $x : W(x, s, c, w, y)$
  then
   ...
end
```

The sequent takes as hypothese the axioms and theorems, the abstract and concrete invariants, the concrete event guards, and the witness predicates for parameters, and one needs to prove that the guard of the event of the abstract machine holds

Axioms and theorems	$A(s, c)$
Abstract invariants	$I(s, c, v)$
Concrete invariants	$J(s, c, v, w)$
Concrete Event Guards	$H(y, s, c, w)$
Witness predicates for parameters	$W(x, s, c, w, y)$
$\vdash$	$\vdash$
Abstract event specific guard	$g(s, c, v, x)$

For the example presented in Figures 6.1 and 6.2, Rodin generates a "*search*/**grd2**/GRD" proof obligation that states that the guards of the concrete event *search* are stronger than the guard **grd2** in the abstract event *search*:

$$
\begin{array}{l}
\quad n \in \mathbb{N} \\
\quad f \in 1\,..\,n \rightarrow \mathsf{D} \\
\quad v \in ran(f) \\
\quad n \in \mathbb{N}1 \\
\quad i \in 1\,..\,n \\
\quad j \in 0\,..\,n \\
\quad v \notin f[i\,..\,j] \\
\quad v \in f[j+1\,..\,n] \\
\quad f(j+1) = v \\
\quad j+1 = k \\
\vdash \\
\quad f(k) = v
\end{array}
$$

The Simulation Proof Obligation: it states that the execution of a concrete event *evt* is not contradictory with the execution of the abstract event that *evt* is refining. Rodin generates a "*evt*/**act**/SIM" proof obligation that states that the execution of the actions of a concrete event *evt* does not contradict the execution of the action **act** from the *evt*0 abstract event. The follows illustrates an event *evt*0 and its refinement *evt*

```
evt0
  any $x$
  where
    ...
  then
    act  $v :| BA1(s, c, v, x, v')$
end
```

```
evt refines evt0
  any $y$
  where
    $H(y, s, c, w)$
  with
    $x : W1(x, s, c, w, y, w')$
    $v' : W2(v', s, c, w, y, w')$
  then
    act2  $w :| BA2(s, c, w, y, w')$
end
```

The sequent takes as hypothese the axioms and theorems, the abstract and concrete invariants, the concrete event guards, the witness predicates for variables and parameters, and the concrete before-after predicate of the abstract machine, and one needs to prove that the before-after predicate of the abstract event holds

Axioms and theorems	$A(s, c)$
Abstract invariants	$I(s, c, v)$
Concrete invariants	$J(s, c, v, w)$
Concrete Event Guards	$H(y, s, c, w)$
Witness predicates for parameters	$W1(x, s, c, w, y, w')$
Witness predicates for variables	$W2(v', s, c, w, y, w')$
Concrete before − after predicate	$BA2(s, c, w, y, w')$
$\vdash$	$\vdash$
Abstract before − after predicate	$BA1(s, c, x, v')$

For the example presented in Figures 6.1 and 6.2, Rodin generates a "*search*/**act1**/SIM" proof obligations :

$$
\begin{array}{l}
\quad n \in \mathbb{N} \\
\quad f \in 1 \mathinner{..} n \rightarrow \mathsf{D} \\
\quad v \in ran(f) \\
\quad n \in \mathbb{N}1 \\
\quad i \in 1 \mathinner{..} n \\
\quad j \in 0 \mathinner{..} n \\
\quad v \notin f[i \mathinner{..} j] \\
\quad v \in f[j+1 \mathinner{..} n] \\
\quad f(j+1) = v \\
\quad j+1 = k \\
\vdash \\
\quad k = j+1
\end{array}
$$

The numeric variant Proof Obligation: it states that under the guards of each convergent or anticipated event, the numeric variant is a natural number. Rodin generates a "*evt*/NAT" proof obligation that states that under the guards of event *evt*, the variant is a natural number. Let machine *m* and event *evt* be as follows

```
machine $m$
 variables
  $v$
 invariants
  $I(sc, v)$
 variant
  $n(sc, v)$
 events
  ...
 end
end
```

```
$evt$
  status convergent
  any $x$
  where
    $G(s, c, v, x)$
  then
    ...
end
```

The sequent takes as hypotheses the axioms and theorems, the invariant, and the guards of the event, and one needs to prove that the variant is a natural number

Axioms and theorems	$A(s, c)$
Invariants	$I(s, c, v)$
Guards of the event	$G(s, c, v, x)$
$\vdash$	$\vdash$
A numeric variant is a natural number	$n(s, c, v') \in \mathbb{N}$

For the example presented in Figures 6.1 and 6.2 Rodin generates a *progress*/NAT proof obligation

$$
\begin{array}{l}
\quad n \in \mathbb{N} \\
\quad f \in 1\,..\,n \rightarrow \mathsf{D} \\
\quad v \in ran(f) \\
\quad n \in \mathbb{N}1 \\
\quad i \in 1\,..\,n \\
\quad j \in 0\,..\,n \\
\quad v \notin f[i\,..\,j] \\
\quad v \in f[j+1\,..\,n] \\
\quad f(j+1) \neq v \\
\vdash \\
\quad n - j \in \mathbb{N}
\end{array}
$$

The Variant Proof Obligation: it states that each convergent event decreases the numeric variant, and each anticipated event does not increase the numeric variant. Rodin generates a "*evt*/VAR" proof obligation that states that a convergent event *evt* decreases the variant, also that an anticipated event *evt* does not increase the variant. Let *m* be a machine and *evt* an event as follows

```
machine m
 variables
  v
 invariants
  I(sc, v)
 variant
  n(sc, v)
 events
  ...
 end
end
```

```
evt
  status convergent
  any x
  where
    G(s, c, v, x)
  then
    ...
end
```

The sequent takes as hypotheses the axioms and theorems, the invariant, the guards of the event, and the before-after predicate of the event, and one needs to prove that the value of the modified variant is smaller than the value of variant without modification

Axioms and theorems	$A(s, c)$
Invariants	$I(s, c, v)$
Guards of the event	$G(s, c, v, x)$
Before − after predicate of the event	$BA(s, c, v, x, v')$
$\vdash$	$\vdash$
Modified variant smaller than variant	$n(s, c, v') < n(s, c, v)$

For anticipated events, one needs to prove that the modified variant is not greater than the variant $(\ldots \vdash n(s, c, v') \leq n(s, c, v))$. For the example presented in Figures 6.1 and 6.2 Rodin generates a *progress*/VAR proof obligation

$$
\begin{array}{l}
\quad n \in \mathbb{N} \\
\quad f \in 1 \,..\, n \rightarrow \mathsf{D} \\
\quad v \in ran(f) \\
\quad n \in \mathbb{N}1 \\
\quad i \in 1 \,..\, n \\
\quad j \in 0 \,..\, n \\
\quad v \notin f[i \,..\, j] \\
\quad v \in f[j+1 \,..\, n] \\
\quad f(j+1) \neq v \\
\quad j' = j+1 \\
\vdash \\
\quad n-(j+1) < n-j
\end{array}
$$

The non-deterministic witness Proof Obligation: it states that each witness of a concrete event indeed exists. Rodin generates a "$evt/x/$WFIS" proof obligation that states that a witness x of a concrete event evt exists. Let evt be a concrete event

$$
\begin{array}{l}
evt \ \mathsf{refines} \ evt0 \\
\quad \mathsf{any} \ y \\
\quad \mathsf{where} \\
\qquad H(y,s,c,w) \\
\quad \mathsf{with} \\
\qquad x : W(x,s,c,w,y,w') \\
\quad \mathsf{then} \\
\qquad BA2(s,c,w,y,w') \\
\mathsf{end}
\end{array}
$$

The sequent takes as hypotheses the axioms and theorems, the abstract and concrete invariants, the concrete event guards, and the concrete before-after predicate, and one needs to prove that the witness exists

$$
\begin{array}{ll}
\textit{Axioms and theorems} & A(s,c) \\
\textit{Abstract invariants} & I(s,c,v) \\
\textit{Concrete invariants} & J(s,c,v,w) \\
\textit{Concrete Event Guards} & H(y,s,c,w) \\
\textit{Concrete before}-\textit{after predicate} & BA2(s,c,w,y,w') \\
\vdash & \vdash \\
\textit{Existance of witness} & \exists\, x \cdot W(x,s,c,w,y,w')
\end{array}
$$

For the example presented in Figures 6.1 and 6.2, Rodin generates a "$search/k/$WFIS" proof obligation

$$
\begin{array}{l}
\quad n \in \mathbb{N} \\
\quad f \in 1\,..\,n \rightarrow \mathsf{D} \\
\quad v \in \mathit{ran}(f) \\
\quad n \in \mathbb{N}1 \\
\quad i \in 1\,..\,n \\
\quad j \in 0\,..\,n \\
\quad v \notin f[i\,..\,j] \\
\quad v \in f[j+1\,..\,n] \\
\quad f(j+1) = v \\
\quad i' = j+1 \\
\vdash \\
\quad \exists\, k \cdot j+1 = k
\end{array}
$$

The theorem Proof Obligation: it states that a context or a machine theorem is indeed provable. For the example presented in Figure 6.1, Rodin generates a "**thm1**/THM" proof obligation to prove that the theorem **thm1** is indeed provable

$$
\begin{array}{l}
\quad n \in \mathbb{N} \\
\quad f \in 1\,..\,n \rightarrow \mathsf{D} \\
\quad v \in \mathit{ran}(f) \\
\vdash \\
\quad n \in \mathbb{N}1
\end{array}
$$

The Well-Definedness Proof Obligation: it states that axioms, theorems, invariants, guards, actions, variants, and witnesses are well defined. For the example presented in Figure 6.1, Rodin generates a "*search*/**grd2**/WD" proof obligation that states that guard **grd2** of event *search* is well defined:

$$
\begin{array}{l}
\quad n \in \mathbb{N} \\
\quad f \in 1\,..\,n \rightarrow \mathsf{D} \\
\quad v \in \mathit{ran}(f) \\
\quad n \in \mathbb{N}1 \\
\quad i \in 1\,..\,n \\
\quad k \in 1\,..\,n \\
\vdash \\
\quad k \in \mathit{dom}(f) \wedge f \in \mathbb{Z} \mathrel{⇸} \mathsf{D}
\end{array}
$$

The Feasibility Proof Obligation: it states that a non-deterministic action is feasible. Rodin generates a "*evt*/**act**/FIS" proof obligation that states that the non-deterministic assignment in action **act** of event *evt* is feasible. Let *evt* be an event

```
$evt$
  any $x$
  where
    $G(s, c, v, x)$
  then
    **act** $v :\mid BA(s, c, v, x, v')$
end
```

The sequent takes as hypotheses the axioms and theorems, the invariants, and the event guards, and one needs to prove the existence of v'

Axioms and theorems	$A(s, c)$
Invariants	$I(s, c, v)$
Event Guards	$G(s, c, v, x)$
$\vdash$	$\vdash$
$\exists\, v' \cdot before - after\ predicate$	$\exists\, v' \cdot BA(s, c, v, x, w')$

The Guard Merging Proof Obligation: it states that the guards of a concrete event that merges two abstract events are stronger that the disjunction abstract events' guards. Rodin generates a "*evt*/MRG" proof obligation that states that the *evt*'s guards are stronger than the disjunction of the abstract events *evt* is refining. Let *evt* be an event that refines both *evt*01 and *evt*02 abstract events

```
$evt01$
  any $x$
  where
    $G1(s, c, v, x)$
  then
    $S$
end
```

```
$evt02$
  any $x$
  where
    $G2(s, c, v, x)$
  then
    $S$
end
```

```
$evt$ refines
    $evt01$
    $evt02$
  any $x$
  where
    $H(s, c, v, x)$
  then
    $S$
end
```

The sequent takes as hypotheses the axioms and theorems, the abstract invariants, and the concrete event guards, and one needs to prove the disjunction of the abstract guards

Axioms and theorems	$A(s, c)$
Abstract Invariants	$I(s, c, v)$
Concrete Event Guards	$H(s, c, v, x)$
$\vdash$	$\vdash$
Disjunction of abstract guards	$G1(s, c, v, x) \vee G2(s, c, v, x)$

Next section shows how each proof obligation generated by Rodin is translated to the input language of Dafny.

```
datatype Pair<S,T> = Pr(x: S, y: T);

datatype Relation<D,R> =
  Rel(domain: set<D>, range: set<R>,
       map: set<Pair<D,R>>);
```

Figure 6.3: Formalising relation structures in Dafny.

6.2 Expressing Event-B Proof Obligations in Dafny

Dafny programming language does not natively support all structures for predicate and set theory used in Event-B. Sets, relations, and other Event-B structures were defined in Dafny as datatypes, and operations over these structures as functions. A relation `Relation<D,R>` between a set of type `D` and a set of type `R` is a constructed type in Dafny, formalised with the aid of a `Rel` type constructor, which has three parameters, `domain` of type `D`, `range` of type `R`, and a `map` between the domain and the range, formalised as a set of pairs (as depicted in Figure 6.3). For the `Rel` type constructor, in the style of other languages like Objective Caml or the PVS language [85], Dafny implicitly declares a `Rel?` predicate that returns **`true`** of any constructed element formed with the type constructor `Rel`.

Modelling relations and sets as datatypes rather than as classes has two main advantages in Dafny. First, instances of classes require new allocations, and second, their fields would need method frame declarations (the **`modifies`** clause of Dafny), which can degrade the performance of Dafny/Boogie/Z3. However, note that an Event-B relation can be used anywhere that a set can appear (a relation is a set of pairs), but unfortunately datatypes cannot be inherited. Therefore, the translation from Event-B to Dafny makes sure that operations are called on the right datatype.

The following sections present the translation of contexts and machines, proof obligations and Event-B operators.

6.2.1 Translating Event-B machines

It is necessary to translate Event-B machine and context information to Dafny for Dafny to be able to discharge the proof obligations: definition of carriers sets, constants, and variables are necessary since sequents use them and they need to be defined in Dafny; definition of axioms, theorems, and invariants are also necessary since sequents are composed of them as explained in the previous section. The translation of Event-B machines to Dafny uses operators that were defined via syntactic rules. Dafny operator translates Event-B machines and contexts to Dafny programming language. Dafny is helped by operator TypeOf which translates the type of Event-B variables and constants to the

$$\frac{\begin{array}{ll}\mathsf{Dafny}(\mathsf{constants}\ c) = \mathtt{C} & \mathsf{Dafny}(\mathsf{sets}\ S) = \mathtt{S}\\ \mathsf{Dafny}(\mathsf{axioms}\ X(s,c)) = \mathtt{X} & \mathsf{Dafny}(\mathsf{theorems}\ T(s,c)) = \mathtt{T}\\ \mathsf{Dafny}(\mathsf{invariants}\ I(s,c,v)) = \mathtt{I} & \mathsf{Dafny}(\mathsf{variables}\ v) = \mathtt{V}\end{array}}{\begin{array}{l}\mathsf{Dafny}(\\ \quad \begin{array}{l}\mathsf{machine}\ M\\ \quad \mathsf{sees}\ ctx\\ \quad \mathsf{variables}\ v\\ \quad \mathsf{invariants}\ I(s,c,v)\\ \quad \mathsf{events}\ e\\ \mathsf{end}\end{array} \qquad \begin{array}{l}\mathsf{context}\ ctx\\ \quad \mathsf{sets}\ S\\ \quad \mathsf{constants}\ c\\ \quad \mathsf{axioms}\ X(s,c)\\ \quad \mathsf{theorems}\ T(s,c)\\ \mathsf{end}\end{array}\\ ) =\\ \mathtt{S}\quad \mathtt{C}\quad \mathtt{X}\\ \\ \mathtt{T}\quad \mathtt{V}\quad \mathtt{I}\end{array}}\ (\mathsf{Prel})$$

Figure 6.4: Translation rule for an Event-B machine and its context to Dafny

corresponding type in Dafny, using the datatypes explained above. It also uses the operator Pred to translate any Event-B predicate or expression to Dafny.

Figure 6.4 depicts the syntactic rule Prel to translate Event-B machine M and the context it sees to Dafny. This information is necessary to discharge the proof obligations. Translation of refinement machines follows the same structures as rule M adding the gluing invariant and the new variables.

Carrier sets are being modelled as set of integers.

$$\frac{}{\begin{array}{c}\mathsf{Dafny}(\mathsf{Sets}\ S) =\\ \mathbf{var}\, S : \mathtt{Set<Integer>;}\end{array}}\ (\mathsf{Sets})$$

As Dafny does not include constants or axioms, constants are being modelled as 0-ary integer functions and axioms as boolean functions with a post-condition that introduces the axiom. Constants are assumed in the translation of a proof obligation. Theorems are translated similar to axioms, but they are checked (the clause **`assert`**) instead. The operator TypeOf in rules Constants and Var returns the corresponding Dafny datatype type as explained at the beginning of this section (Section 6.2)

$$\frac{\mathsf{TypeOf}(c) = \mathtt{Type}}{\begin{array}{c}\mathsf{Dafny}(\mathsf{constants}\ c) =\\ \mathbf{function}\, c() : \mathtt{Type;}\end{array}}\ (\mathsf{Constants})$$

$$\frac{\mathsf{Pred}(A) = \mathtt{A}}{\begin{array}{l}\mathsf{Dafny}(\mathsf{axioms}\ A) =\\ \mathbf{function}\, axm() : \mathtt{bool}\\ \qquad \mathbf{ensures}\, \mathtt{A}\end{array}}\ (\mathsf{Axioms})$$

Event-B variables are modelled as variables in Dafny

$$\frac{\mathsf{TypeOf}(v) = \texttt{Type}}{\begin{array}{l}\mathsf{Dafny}(\mathsf{variables}\ v) = \\ \quad \texttt{\textbf{var}}\, v : \texttt{Type;}\end{array}}\ (\mathsf{Var})$$

Invariants are modelled as boolean functions with postconditions that can be assumed or asserted. Event-B variables, constants, and carrier sets are implicitly non-null. This non-nullness condition has to be made explicit in Dafny and assumed by all machine invariants. In Rule Invariants, the operator DafnyVar collects all variables from the invariant.

$$\frac{\mathsf{Pred}(I) = \texttt{I} \quad \mathsf{DafnyVar}(I) = \texttt{setVar}}{\begin{array}{l}\mathsf{Dafny}(\mathsf{Invariants}\ I) = \\ \texttt{\textbf{function}}\, \mathit{nonNullnessCond}() : \texttt{bool}\{ \\ \quad \texttt{setVar != } \texttt{\textbf{null}} \\ \} \\ \\ \texttt{\textbf{function}}\ \mathit{inv}() :\ \texttt{bool;} \\ \quad \texttt{\textbf{requires}}\, \mathit{nonNullnessCond}()\texttt{;} \\ \quad \texttt{\textbf{ensures}}\, \texttt{I;}\end{array}}\ (\mathsf{Invariants})$$

6.2.2 Translating Event-B proof obligations

The translation of Event-B proof obligations to Dafny uses operators that were defined via syntactic rules, one for each proof obligation generated by Rodin (e.g. INV, MRG, GRD, Sim, NAT, FIS, WFIS, VAR). The parameters for each operator depend on the type of the PO, for instance, the operator FIS for the feasibility proof obligation takes one parameter, an Event-B machine, whereas the operator GRD takes two, an Event-B machine and its refinement. Depending on the type of proof obligation generated by Rodin, a **`ghost method`** is declared that might or might not assume local invariants, theorems or axioms. An additional operator (Ctx) is defined that translates Event-B axioms and theorems presented in a context to Dafny. The rules are presented as follows:

The Invariant Preservation Proof Obligation: rule INV generates a method in Dafny that assumes the translation of the invariants of the abstract and concrete machines, the translation non-nullness axiom and theorems, the translation of the predicate related to the witness, the translation of the before-after predicate of the concrete event, and the translation of the guards of the refined event. The method finally asserts the result of the translation of the modified invariant.

```
Ctx(Ctx) = AT
Dafny(invariant I(s, c, v)) = I        Dafny(invariant J(s, c, v, w)) = J
Pred(H(y, s, c, w)) = H                Pred( W2(v', s, c, w, y, w')) = W2
Pred(BA2(s, c, w, y, w')) = BA2        Dafny(invariant J(s, c, v', w')) = J'
─────────────────────────────────────────────────────────────────────────── (Inv)
    Inv_Concrete(
                                     machine N refines M
                                     variables w
        machine M sees Ctx           invariant J(s, c, v, w)
        variables v                  event evt refines evt0
        invariant I(s, c, v)            any y where
        event evt0                         H(y, s, c, w)
           . . .                        with
           end                             v' :| W2(v', s, c, w, y, w')
        end ,                           then
                                           w :| BA2(s, c, w, y, w')
                                        end
                                     end )
    =
     ghost method evt_inv_INV()
      assume AT && I && J && H && W2 && BA2;
      assert J';
```

Rule Inv also takes into account the invariant preservation proof obligation for just an abstract machine. Rule Inv showed bellow generates a Dafny method that assumes the translation of the abstract invariant, the translation of the non-nullness axiom and theorems, the translation of the before-after predicate and guards of the abstract event. The method finally asserts the result of the translation of the modified invariant.

```
Ctx(Ctx) = AT
Dafny(invariant I(s, c, v)) = I     Pred(G(s, c, v, x)) = G
Pred(BA(s, c, v, x, v')) = BA       Dafny(invariant I(s, c, v')) = I'
───────────────────────────────────────────────────────────────── (Inv)
                 INV_Abstract(
                 machine M sees Ctx
                 variables v
                 invariant I(s, c, v)
                 event evt
                    any x where
                       G(s, c, v, x)
                    then
                       v :| BA(s, c, v, x, v')
                    end
                 end )
                 =
                 ghost method evt_inv_INV()
                  assume AT && I && G && BA;
                  assert I';
```

The Feasibility Proof Obligation: rule Feas generates a Dafny method that assumes the translation of the invariants, the translation of the non-nullness axioms and theorems, and the translation of the guard of the event. The method finally asserts the result of the translation of the existence of the witness value ensuring the before-after predicate of the event.

```
Ctx($Ctx$) = AT               Dafny(invariants $I(s,c,v)$) = I
Pred($G(s,c,v,x)$) = G        Pred($BA(s,c,v,x,v')$) = BA
---------------------------------------------------------------- (Feas)
                FIS(
                machine $M$ sees $Ctx$
                variables $v$
                invariant $I(s,c,v)$
                event $evt$
                  any $x$ where
                    $G(s,c,v,x)$
                  then
                    $v$ :| $BA(s,c,v,x,v')$
                  end
                end ) =
                ghost method $evt\_act\_FIS()$
                 assume AT && I && G;
                 assert  (exists $v'$:: BA);
```

The Guard Strengthening Proof Obligation: rule Grd generates a method in Dafny that assumes the translation of the abstract and concrete invariants, the translation of the non-nullness axioms and theorems, the translation of the guard of the refined event, and the translation of the predicate related to the witness. The method finally asserts the result of the translation of the guard of the abstract event.

```
 Dafny(invariants $I(s,c,v)$) = I         Ctx($Ctx$) = AT
 Dafny(invariant $J(s,c,v,w)$) = J        Pred($H(y,s,c,w)$) = H
 Pred($W(x,s,c,w,y)$) = W                 Pred($g(s,c,v,x)$) = G
---------------------------------------------------------------------- (Grd)
GRD(
  machine $M$ sees $Ctx$          machine $N$ refines $M$
  variables $v$                   variables $w$
  invariant $I(s,c,v)$            invariant $J(s,c,v,w)$
  event $evt0$                    event $evt$ refines $evt0$
    any $x$ where                   any $y$ where
      grd $g(s,c,v,x)$                $H(y,s,c,w)$
    then                            with      $x$ :| $W(x,s,c,w,y)$
      ...                           then      ...
    end                             end
  end ,                           end )
=
 ghost method $evt\_grd\_GRD()$
  assume AT && I && J && H && W;
  assert G;
```

The Guard Merging Proof Obligation: rule MRG generates a method in Dafny that assumes the translation of the abstract invariant, the translation of the non-nullness axioms and theorems, and the translation of the guard of the concrete event. The method finally asserts the result of the translation of the disjunction of the guards of the abstract events being merged.

```
Ctx(Ctx) = AT
Dafny(invariants I(s, c, v)) = I    Pred(H(s, c, v, x)) = H
Pred(G1(s, c, v, x)) = G1           Pred(G2(s, c, v, x)) = G2
----------------------------------------------------------------- (MRG)
  MRG(
    machine M sees Ctx
    variables v
    invariant I(s, c, v)         machine N refines M
    event evt01                  variables w
      any x where                invariant J(s, c, v, w)
        G1(s, c, v, x)           event evt refines
      then                             evt01
        S                              evt02
      end                          any x where
                                     H(s, c, v, x)
    event evt02                    then
      any x where                    S
        G2(s, c, v, x)             end
      then                       end )
        S
      end
    end ,
  =
   ghost method evt_MRG()
    assume AT && I && H;
    assert G1 ∨ G2;
```

The Simulation Proof Obligation: rule Sim generates a Dafny method that assumes the translation of the abstract and concrete invariants, the translation of the non-nullness axioms and theorems, the translation of the guard of the concrete event, the translation of the predicate related to the witness, and the translation of the before-after predicate of the concrete event. The method finally asserts the result of the translation of the before-after predicate of the abstract event.

$$
\begin{array}{ll}
\mathsf{Dafny}(\mathsf{invariants}\ I(s,c,v)) = \mathtt{I} & \mathsf{Ctx}(Ctx) = \mathtt{AT} \\
\mathsf{Dafny}(\mathsf{invariants}\ J(s,c,v,w)) = \mathtt{J} & \mathsf{Pred}(H(y,s,c,w)) = \mathtt{H} \\
\mathsf{Pred}(W1(x,s,c,w,y,w')) = \mathtt{W1} & \mathsf{Pred}(BA2(s,c,w,y,w)) = \mathtt{BA2} \\
\mathsf{Pred}(W2(v',s,c,w,y,w')) = \mathtt{W2} & \mathsf{Pred}(BA1(s,c,v,x,v')) = \mathtt{BA1} \\
\hline
\end{array}
\ (\mathsf{Sim})
$$

```
Sim(
   machine $M$
    sees $Ctx$
   variables $v$
   invariant $I(s, c, v)$
   event $evt0$
      any $x$
    where
         ...
      then
         $v$ :| $BA1(s, c, v, x, v')$
      end
   end ,

   machine $N$
    refines $M$
   variables $w$
   invariant $J(s, c, v, w)$
   event $evt$
    refines $evt0$
      any $y$
    where
         $H(y, s, c, w)$
      with
         $x$ :| $W1(x, s, c, w, y, w')$
         $v'$ :| $W2(v', s, c, w, y, w')$
      then
         $w$ :| $BA2(s, c, w, y, w')$
      end
   end )
=
 ghost method evt_act_SIM()
  assume AT && I && J && H && W1 && W2 && BA2;
  assert BA1;
```

The Numeric Variant Proof Obligation: rule Nat generates a Dafny method assumes the translation of the machine invariant, the translation of the non-nullness axioms and theorems, and the translation of the guard of the event. The method finally asserts the result of the translation when the evaluation of the variant is a natural number. In rule Nat, *Nat* is defined as set in Dafny that contains natural numbers. Dafny

```
Dafny(invariants $I(s,c,v)$) = I    Ctx($Ctx$) = AT
Pred($G(s,c,v,x)$) = G              Pred($n(s,c,v)$) = n
------------------------------------------------------- (Nat)
          NAT(
          machine $M$ sees $Ctx$
          refines ...
          variables $v$
          invariant $I(s,c,v)$
          variant $n(s,c,v)$
          event $evt$
            status convergent //$or$ anticipated
            any $x$ where
              $G(s,c,v,x)$
            then
              ...
            end
          end ) =
          ghost method $evt\_NAT()$
           assume AT && I && G;
           assert $Nat$.has(n);
```

The Variant (VAR): There exist two different proof obligations related to the variant. It regards on the status of the event (i.e. convergent or anticipated). The following is the rule translation for variant proof for a convergent event, the method asserts the result of the translation when the evaluation of the variant with the new values of variables is lower than the previous evaluation

```
Dafny(invariants $I(s,c,v)$) = I    Ctx($Ctx$) = C
Pred($G(s,c,v,x)$) = G              Pred($BA(s,c,v,x,v')$) = BA
Pred($n(s,c,v)$) = n                Pred($n(s,c,v')$) = n'
------------------------------------------------------------- (Conv)
               VAR_Conv(
               machine $M$ sees $Ctx$
               variables $v$
               invariant $I(s,c,v)$
               variant $n(s,c,v)$
               event $evt$
               status convergent
                 any $x$ where
                   $G(s,c,v,x)$
                 then
                   $v$ :| $BA(s,c,v,x,v')$
                 end
               end ) =
               ghost method $evt\_VAR()$
                assume AT && I && G && BA;
                assert n' < n;
```

For an anticipated status in an event, the method asserts the result of the translation when the evaluation of the variant with the new values of variables is lower than equal the previous evaluation

$$\dfrac{\begin{array}{ll} \mathsf{Dafny}(\mathsf{invariants}\ I(s,c,v)) = \mathtt{I} & \mathsf{Ctx}(Ctx) = \mathtt{C} \\ \mathsf{Pred}(G(s,c,v,x)) = \mathtt{G} & \mathsf{Pred}(BA(s,c,v,x,v')) = \mathtt{BA} \\ \mathsf{Pred}(n(s,c,v)) = \mathtt{n} & \mathsf{Pred}(n(s,c,v')) = \mathtt{n'} \end{array}}{\begin{array}{l} \mathsf{VAR\_Ant(} \\ \mathsf{machine}\ M\ \mathsf{sees}\ Ctx \\ \mathsf{variables}\ v \\ \mathsf{invariant}\ I(s,c,v) \\ \mathsf{variant}\ n(s,c,v) \\ \mathsf{event}\ evt \\ \mathsf{status\ anticipated} \\ \quad \mathsf{any}\ x\ \mathsf{where} \\ \quad\quad G(s,c,v,x) \\ \quad \mathsf{then} \\ \quad\quad v :\mid BA(s,c,v,x,v') \\ \quad \mathsf{end} \\ \mathsf{end}\ ) = \\ \textbf{ghost method}\ evt\_VAR() \\ \ \textbf{assume}\ \mathtt{AT} \,\&\&\, \mathtt{I} \,\&\&\, \mathtt{G} \,\&\&\, \mathtt{BA}; \\ \ \textbf{assert}\ \mathtt{n'} \leq \mathtt{n}; \end{array}} \ (\mathsf{Ant})$$

The non-deterministic witness (WFIS): The translated method assumes the invariants of the abstract and concrete machines, the non-nullness axioms and theorems, the guard of the refined event, and the before-after predicate of the refined event. The method finally asserts the result of the translation of the existence of a value x that ensures the witness' predicate

$$\dfrac{\begin{array}{ll} \mathsf{Dafny}(\mathsf{invariants}\ I(s,c,v)) = \mathtt{I} & \mathsf{Ctx}(Ctx) = \mathtt{AT} \\ \mathsf{Dafny}(\mathsf{invariants}\ J(s,c,v,w)) = \mathtt{J} & \mathsf{Pred}(H(y,s,c,w)) = \mathtt{H} \\ \mathsf{Pred}(BA2(s,c,w,y,w')) = \mathtt{BA2} & \mathsf{Pred}(W(x,s,c,w,y,w')) = \mathtt{W} \end{array}}{\begin{array}{l} \mathsf{WFIS(} \\ \quad \begin{array}{ll} \begin{array}{l} \mathsf{machine}\ M\ \mathsf{sees}\ Ctx \\ \mathsf{variables}\ v \\ \mathsf{invariant}\ I(s,c,v) \\ \mathsf{event}\ evt0 \\ \quad\quad \ldots \\ \quad \mathsf{end} \\ \mathsf{end}\ , \end{array} & \begin{array}{l} \mathsf{machine}\ N\ \mathsf{refines}\ M \\ \mathsf{variables}\ w \\ \mathsf{invariant}\ J(s,c,v,w) \\ \mathsf{event}\ evt\ \mathsf{refines}\ evt0 \\ \quad \mathsf{any}\ y\ \mathsf{where} \\ \quad\quad H(y,s,c,w) \\ \quad \mathsf{with} \\ \quad\quad x :\mid W(x,s,c,w,y,w') \\ \quad \mathsf{then} \\ \quad\quad w :\mid BA2(s,c,w,y,w') \\ \quad \mathsf{end} \\ \mathsf{end}\ ) \end{array} \end{array} \\ = \\ \textbf{ghost method}\ evt\_witness\_WFIS() \\ \ \textbf{assume}\ \mathtt{AT} \,\&\&\, \mathtt{I} \,\&\&\, \mathtt{J} \,\&\&\, \mathtt{H} \,\&\&\, \mathtt{BA2}; \\ \ \textbf{assert}\ (\textbf{exists}\, x :: \mathtt{W}); \end{array}} \ (\mathsf{With})$$

6.3 The EventB2Dafny Tool

The EventB2Dafny tool is integrated to Rodin as an Eclipse plug-in. Full source code for EventB2Dafny is available in [30]. The EventB2Dafny tool parses Rodin proof obligations into a Dafny program. The proof obligation can include information about a machine context, e.g. sets and axioms, or machine variables and invariants, which might be conjoined with machine variables and invariants from a refinement machine. EventB2Dafny directly works on proof obligations as generated by Rodin, so the soundness of EventB2Dafny directly lies on the soundness of the proof-generation mechanism backing the Rodin platform.

Figure 6.6 shows a partial output of applying EventB2Dafny to the Event-B model depicted in Figure 6.5. The input is the invariant preservation proof obligation generated by Rodin for the Event-B model showed in Figure 6.5 (page 97) regarding *search* event with respect to invariant **inv1** : $i \in 1\,..\,n$.

$$\begin{array}{ll}
 & n \in \mathbb{N}1 \\
 & f \in 1\,..\,n \rightarrow \mathsf{D} \\
 & v \in ran(f) \\
 & i \in 1\,..\,n \\
 & k \in 1\,..\,n \\
 & f(k) = v \\
\vdash & \\
 & k \in 1\,..\,n
\end{array}$$

Figure 6.5: Proof Obligation generated by Rodin

The user has to prove that given axioms, theorems, invariants, and guards of the event (in this case *search*) the invariant **inv1** holds after the action **act1** ($i := k$).

EventB2Dafny defines the carrier set D as a set of integers. Constants are defined as 0-ary Integer functions. Axioms are translated as 0-ary boolean functions where the axiom is taken as a post-condition. For instance, axiom **ax2** (defined in Event-B as $f \in 1\,..\,n \rightarrow \mathsf{D}$) is translated as function `ax2` which ensures that the variable f is a total function where its domain is equal to the set of number from 1 to n (denoted by **`upto`**). And its range is equal to `D` (type `Int` is the representation of Integers). The translation also defines the non-nullness condition that stands all variables and constants cannot be null values (this is an implicit condition in Event-B). The invariant **inv1** is translated as a boolean function that requires the non-nullness condition and ensures the translation of the invariant. Finally, a **`ghost`** method is defined (`search_inv1_INV`) that assumes the non-nullness condition, the invariants, and the guard of the event. It asserts on the evaluation of the invariant after the execution of the event (where variables have new values) (i.e. $k \in 1\,..\,n$).

```
var D : Set<Integer>;
function f() : Relation<Integer, Integer>;
function n() : Integer;
function v() : Integer;
var i : Integer;

function ax1() : bool
     ensures Nat.has(n());

function ax2() : bool
     ensures f().isTotalFunction() &&
       f().domain == Int.Init().upto(Integer.Init(1), n()).instance &&
       f().range == D.instance;

function ax3() : bool
     ensures f().range.has(v());

function thm1() : bool
     ensures Nat1.has(n());

function nonNullnessCond() : bool
    ensures i!= null && D!= null && f!= null && n!= null && v!= null;

function inv1() : bool
     requires nonNullnessCond();
     ensures Int.Init().upto(Integer.Init(1), n()).has(i);

ghost method search_inv1_INV(){
     assume nonNullnessCond();
     assume inv1();
     assume ax1();
     assume ax2();
     assume ax3();
     assume thm1();
     assume f().funcImage(k).equals(v());
     assert Int.Init().upto(Integer.Init(1), n()).has(k);
}
```

Figure 6.6: Partial EventB2Dafny output

6.4 Conclusion

In this chapter we presented a serie of translation rules to generate Dafny code from Event-B proof obligations. We also introduced the implementation of the rules as the EventB2Dafny tool which is a Rodin's plug-in. We have validated EventB2Dafny by applying it to an Event-B model.

Users of Event-B could benefit from EventB2Dafny since the tool might help them in the process of discharging proof obligations. Proof obligations for complex systems are complex and difficult to discharge. EventB2Dafny gives additional alternatives in the process of discharging a proof obligation by porting it to Dafny. Once the proof obligation is in Dafny, users can use the automatic provers (e.g. Z3 SMT solver) that come with it to prove the proof obligations.

Discharging proof obligations in Dafny is semi-automatic and relies on Dafnys provers performance. We plan to integrate EventB2Dafny to Dafny so that discharging proof obligations would include the following steps

i) to choose a proof obligation in Event-B, *ii*) to use EventB2Dafny to translate it to Dafny, *iii*) to receive feed-back directly from Dafny in Rodin (EventB2Dafny will automatically run Microsoft Visual Studio to discharge the proof, and feed Rodin with the feed-back that Dafny provides).

Chapter 7

Case Studies

This chapter. Through this thesis we have defined a series of tools to work with different formal methods. The work has ended in the translation of Event-B models to JML-annotated Java code. we have also proposed two different techniques on software development where EventB2Java can be used. This chapter shows two case studies on the use of EventB2Java as part as two different software developments, and a benchmark that compares EventB2Java with two existing tools for generating Java code from Event-B models.

The first case study (see Section 7.1) describes the development of a Social-Event Planner using EventB2Java and the Model-View-Controller design pattern. The Social-Event Planner is an Android [61] application of a planner for social events in which a user can create a social event and invite a list of people to join it. The second case study (see Section 7.2) presents the use of EventB2Java in testing a security-critical access control system modelled in Event-B.

Contributions. The main contributions of this chapter are to show how the EventB2Java tool can be incorporated as part of two different software development, showing that different people from different expertise can work together in the said development. To compare our tool with existing tools for generating Java code from Event-B models.

Related work. related work: A preliminary version of an Event-B model of the Tokeneer ID Station (TIS) is presented in [86]. However, the model is a reduced model of the TIS; it consists of a single abstract machine; no machine refinement was defined and a few proof obligations remained undischarged, so we decided to write our Event-B model of Tokeneer afresh.

In [21], an automatic approach to provide correct testing inputs for parameters associated to axioms is described. We can use this work to extend EventB2Java to automatically assign values for constants in Event-B contexts. The work in [15] further constructs test data sets from formal specifications.

In [32], a strategy called JFly is proposed to evolve informal (written in natural language) software requirements into formal requirements written in

JML. This work can be reused to structure the writing of JUnit tests from a STS document.

7.1 The Social-Event Planner

This section describes the development of a Social-Event Planner following a Model-View-Controller (MVC) design pattern. Sections 7.1.1, 7.1.2, and 7.1.3 describe the implementation of the Model of the system by modelling it in Event-B to then transition to Java code using the EventB2Java tool. Section 7.1.4 describes the implementation of the View and Controller of the system.

7.1.1 Requirement Document for the Social-Event Planner

The Social-Event Planner was modelled as a piece of software that runs over the existing Social Network in [37] (the Social Network model is briefly explained Section 4.3.1). For modelling the Social-Event Planner, We followed the "parachute" software development strategy of Event-B proposed by J.-R Abrial in [2]. We classify the requirements within two categories:

- Those concerned with the functionalities of the application labeled *FUN*.
- Those concerned with the decision making labeled *DEC* (e.g. when a user has to make a decision on either going to a social-event or declining).

The main functionality of the Social-Event Planner application is to allow a user to create a social-event and invite other users to it. A social-event shall consist of the content visible to any invited user. The user creating the social-event might enforce a specific privacy policy over the social-event. Such a privacy policy shall consist of a set of restricted users from which the creator of the social-event wants to keep the social event hidden. The creator of the social-event can allow other invited users to further invite additional users to the social-event. These additional users must not belong to the aforementioned set of restricted users. To illustrate this, a user (*UserA*) within the social network creates an event 'Picnic in the Park'. He invites the lists of users 'Close Friends' and 'Trekking Friends'. He decides he doesn't want any user belonging to the list 'Professors' to be invited. An invite will be sent from *UserA* to every member of 'Close Friends' and 'Trekking Friends' that do not belong to list 'Professors'. He grants the users in list 'Close Friends' the 'Invite' privilege. A user (*UserB*) from the list 'Close Friends' decides to invite the users in the list 'Institute'. The social-event will be shared with (an invite will be sent to) every user in the list 'Institute' that doesn't belong to the list 'Professors'. In the following, we present the requirements of the social-event planner.

The users of the Social Network can create social-events.	FUN-1
The creator of a social-event can associate content with it.	FUN-2

The invited users to the social-event can either ‘Join’ the event, ‘Decline’ the invitation or reply with a ‘Maybe’.

The ‘invited’ users can reply to the social-event.	FUN-3
A reply to an invitation shall be either *1*) Join *2*) Decline *3*) Maybe or the user can choose not to reply.	DEC-1
A user invited to a social-event can swap their reply between Join, Decline or Maybe.	FUN-4
A user who has been invited to a social-event can view all the content associated with the social-event.	FUN-5

The users of the Social Network invited to a social-event can be granted permissions to View or Edit content associated with the social-event. Additionally specific users can be granted permission to invite additional users.

The users shall be able to view or edit a social-event or invite other users to the social-event based on permissions.	FUN-6
The following permissions can be awarded over a social-event to a user on the Social Network: 1) View 2) Edit 3) Invite.	DEC-2

The creator of the event, called the ‘owner’, might allow the invited users to further invite users by explicitly granting them the permission to do so.

The user that creates a social-event shall be designated the owner of the social-event.	FUN-7
The owner of any social-event shall be granted all privileges over it.	DEC-3
A user can invite a list of users to a social-event.	FUN-8
The owner of a social-event can grant ‘Invite’ permissions to any user that has been invited.	FUN-9
A user with an ‘Invite’ permission to a social-event shall be allowed to invite other users to the social-event.	FUN-10
The users that has been invited to a social-event can add content to the social-event in the form of comments.	FUN-11

Refinement Strategy. Below is listed the order in which the various proposed requirements were taken into account.

ref_socialevents. Once we have the social networking core we incorporate the possibility for a user to create social-events and associate content with it. In this direction we take care of requirements FUN-1, FUN-2.

ref_socialinvite. This refinement includes the functionality for the owner of a social-event to invite other users to the social-event. The owner can grant permissions to specific invited users allowing them to invite other users. An invited user can reply to an invitation with Join/Decline or Maybe or change and existing reply. This refinement satisfies requirements FUN-3,FUN-4,FUN-7, FUN-8,FUN-9,FUN-10, FUN-11, DEC-1 and DEC-2.

ref_socialpermissions. Then we add privileges in order for people to view and edit content associated with social-events. It takes into account requirements FUN-5 and FUN-6.

7.1.2 The Event-B Model of the Social-Event Planner

The Social-Event Planner works on top of the Event-B model for Social Networking presented in [38]. The Social Networking specifies a social network as composed of people and content (e.g. photos, videos, comments). People within the social network can share their own content. For that, the model defines permissions over the content. The Social-Event Planner can be regarded as a plug-in of the Social Networking. The Social-Event Planner is composed of three machines ($ref6\_socialevents$, $ref7\_socialinvite$ and $ref8\_socialpermissions$) that constitute refinements of the Social Networking model. The follow explains part of the machines. The full source (the Social Networking and the Social-event Planner) is available at [90].

In the following, we explain the refinements of the Social-Event Planner :

ref6_socialevents: This machine represents the core of the Social-Event Planner. A user in the social network can create social-events and upload information to it. Figure 7.1 depicts part of the Event-B machine.

Machine $ref6\_socialevents$ **see**s context ctx_event depicted in the right of Figure 7.1. The context defines a carrier set EVENTS containing all possible social-events. The machine defines variable *events* representing the actual social-events created within the Social Networking. Variable *scontents* is a set of contents present in the social-event (e.g. a picture or a comment within a specific social-event). Variable *eventcontents* is a relation that maps contents to social-events. This relation allows the system to know which content belongs to which social-event. The relation is defined as a total relation so that a social-event can contain several contents and a content must be in at least one social-events. And variable *eventowner* defined as a total function that maps social-events to person. It models each social-event has an unique owner. Variables *contents*, and *person* were defined in previous refined machine, they describe the contents in the social networking, and the set of actual people in the social network, respectively.

```
machine ref6_socialevents
   refines ref5_lists sees ctx_events
 variables  sevents scontents
   eventcontents eventowner
 invariant
  invr6_1  sevents ⊆ EVENTS
  invr6_2  scontents ⊆ contents
  invr6_3  eventcontents ∈ scontents ↞→ sevents
  invr6_4  eventowner ∈ sevents → persons
 events
  create_social_event
   any pe se
   where
    grdr6_1  pe ∈ persons
    grdr6_2  se ∈ EVENTS\sevents
   then
    actr6_1  sevents := sevents ∪ se
    actr6_2  eventowner(se) := pe
  end
end
  upload_principal_content_planner
    extends upload_principal
   any se where
    grdr6_1  se ∈ sevents
   then
    actr6_1  scontents := scontents ∪ c2
    actr6_2  eventcontents :=
     eventcontents ∪ c2 ↦ se
  end
end
```

```
context ctx_events
  extends ctx_lists
sets EVENTS
end
```

Figure 7.1: Excerpt of the *ref6_socialevents* (and the context it sees) Event-B machine for the Social-Event Planer model

Event *create_social_event* models the creation of a new social-event *se* (not presented already in the social-event Planner) by a user *pe* that belong to the Social Networking. The execution of this event adds the new social-event *se* to the set of existing social-events and defines *pe* as the owner. Event *upload_principal_content_planner* allows users of the social-event to upload a principal content in the social-event (local variable $c2$ correspond to the content and it is defined in the abstract event). Symbols $\mapsto$, $\cup$, $\setminus$, and $\subseteq$ model a pair of elements, set union, set difference, and set subset in Event-B, respectively. The machine defines more events not shown in the figure.

ref7_socialinvite: This machine specifies the users invited to a social-event. Figure 7.2 depicts part of the Event-B refinement. The machine **refines** machine *ref6_socialevents* and **sees** the same context, it models the set of invited people to a social-event by using the variable *invited*, which is a relation that maps social-events to invited people. The variable is defined as a relation, so that, a user can be invited to several social-events and a social-event can contain several invited people. An invited user can reply to an invitation. The user can either 'join' the event, or reply as a 'maybe' or 'decline'. This is represented by variables *join*, *maybe* and *decline* that are modelled as relations that map social-events to person. A user can be in just one of these states. This is ensured by invariants **invr7_5**, **invr7_6**, and **invr7_7**. Invariant **invr7_8** states that a user can reply to one of these states if the user is invited to the social-event. The owner of the social-event and a set of invited people to that social-event have the privilege to invite more people. This is modelled using the variable *populate*.

Invariant **invr7_10** states that the owner of a social-event has the right to invite any one to it. Finally, invariant **invr7_11** states that people with the right to invite other people are invited to the social-event.

The machine also defines the event *sent_invite* allows the invited people to the social-event *se* who are in the *populate* variable (**grdr7_3** $se \mapsto pe \in populate$) to invite more people from a list $l1$. Variable *listpe* is defined in a previous refined machines. It is defined as relation that maps a list identifier to a set of people. Event *grant_populate* grants permission to an invited user to invite more people. Just the owner of the social-event can grant such permission. Events *reply_with_join*, *reply_with_maybe*, and *reply_with_decline* model the possibility to reply a social-event with 'join', 'maybe' or 'decline', respectively (these events are not shown in Figure 7.2). Symbols $\times$, and $\varnothing$ model a cartesian product, and empty set in Event-B, respectively.

ref8_socialpermission: Finally, machine *ref_8socialpermission* specifies the permission over the content involved in a specific social-event. An invited user can have permission to view or edit a specific content. The machine models these permissions by using the variables *socialview*, which is defined as a relation that maps social content within the social-event to person that has privilege to view that content. And the variable *socialedit* which is defined as a relation that maps social content to person that has privilege to edit that content. Figure 7.3 depicts part of the Event-B machine.

The invariant **invr8_3** specifies that any invited person to a social-event has privilege to view the content on it. Invariant **invr8_4** states that the owner of the social-event has privilege to edit the content on it. Finally, invariant **invr8_5** models that who has privilege to edit a content, has also privilege to view it.

This machine does not define any new event, but it extends the previous ones. For instance, when a user *pe* creates a social-event, that user has permission to view and edit the content involved in that social-event. The symbol $\triangleright$ models

```
machine $ref7\_socialinvite$ refines $ref6\_socialevents$ sees $ctx\_events$
 variables $invited\ populate\ join\ maybe\ decline$
 invariant
   invr7_1 $invited \in sevents \leftrightarrow persons$      invr7_2 $join \in sevents \leftrightarrow persons$
   invr7_3 $maybe \in sevents \leftrightarrow persons$        invr7_4 $decline \in sevents \leftrightarrow persons$
   invr7_5 $join \cap maybe = \varnothing$                    invr7_6 $join \cap decline = \varnothing$
   invr7_7 $maybe \cap decline = \varnothing$                 invr7_8 $join \cup maybe \cup decline \subseteq invited$
   invr7_9 $populate \in sevents \leftrightarrow persons$     invr7_10 $eventowner \subseteq populate$
   invr7_11 $populate \subseteq invited$
 events
  $sent\_invitate$
  any $pe\ se\ l1$
  where
   grdr7_1 $l1 \in dom(listpe)$ grdr7_2 $se \in sevents$
   grdr7_3 $se \mapsto pe \in populate$
  then
   actr7_1 $invited := invited \cup (se \times listpe[l1])$
  end
  $grant\_populate$
  any $ow\ pe\ se$
  where
    grdr7_1 $ow \in persons$                 grdr7_2 $se \in sevents$
    grdr7_3 $ow = eventowner(se)$            grdr7_4 $pe \in persons$
    grdr7_5 $se \mapsto pe \in invited$
  then
   actr7_1 $populate := populate \cup se \mapsto pe$
  end
end
```

Figure 7.2: Excerpt of the *ref7_socialinvite* Event-B machine for the Social-Event Planer model

a range restriction in Event-B.

We model the Social-Event Planner in Rodin. All proof obligations generated by Rodin were discharged.

7.1.3 Generating JML-annotated Java code for the Social-Event Planner Event-B model

We used EventB2Java to generate JML-annotated Java code for the last refinement of the Social-Event Planner. The EventB2Java tool generates one Java class (see an excerpt of the Java class in Figure 7.4) containing the translation of the carrier sets, constants and variables (with their respective initialisations), and the Event-B invariant.

The tool also generates a Java `Thread` implementation for each machine event. Figure 7.5 shows the translation of one event: *create_social_event*,

machine $ref8\_socialpermissions$ refines $ref7\_socialinvite$ sees ctx_events
variables
$socialviewp$ $socialeditp$
invariant
invr8_1 $socialviewp \in scontents \leftrightarrow persons$
invr8_2 $socialeditp \in scontents \leftrightarrow persons$
invr8_3 $eventcontents;\ invited \subseteq socialviewp$
invr8_4 $eventcontents;\ eventowner \subseteq socialeditp$
invr8_5 $socialeditp \subseteq socialviewp$
events
$create\_social\_event$ extends $create\_social\_event$
then
actr8_1 $socialviewp := socialviewp \cup (dom(eventcontents \rhd se) \times pe)$
actr8_2 $socialeditp := socialeditp \cup (dom(eventcontents \rhd se) \times pe)$
end
end

Figure 7.3: Excerpt of the *ref8_socialpermission* Event-B machine for the Social-Event Planer model

where `m` is a reference to the machine class implementation (used to access machine variables via getter and setter methods). Methods `guard_create_social_event` and `run_create_social_event` implement the behaviour of the *create_social_event* event in Java. The first method checks the event guard, and the second may execute when that guard holds. Whether `run_create_social_event` executes when `guard_create_social_event` holds is determined by the `run()` method of `create_social_event` in coordination with the respective `run()` methods of all existing events.

Variables `contents_tmp`, `pages_tmp`, . . . hold temporary values of variables `contents`, `pages`, . . . , respectively. EventB2Java uses these temporary values to implement simultaneous assignment in Java.

The JML-annotated Java code generated by EventB2Java from the last refinement of the Social-Event Planner Event-B model represents the Model (M) of a MVC design pattern development. We extended this core functionality to implement a usable version of the Social-Event Planner as an Android application.

7.1.4 The View and Controller Parts of the Social-Event Planner

The View and Controller part of the system were developed in Java using the Android API. The View part allows users to interact with the Social-Event Planner. The *Controller* part makes a bridge between the View part and the Model. Figure 7.6 depicts two of the main screen shots of the user interface for

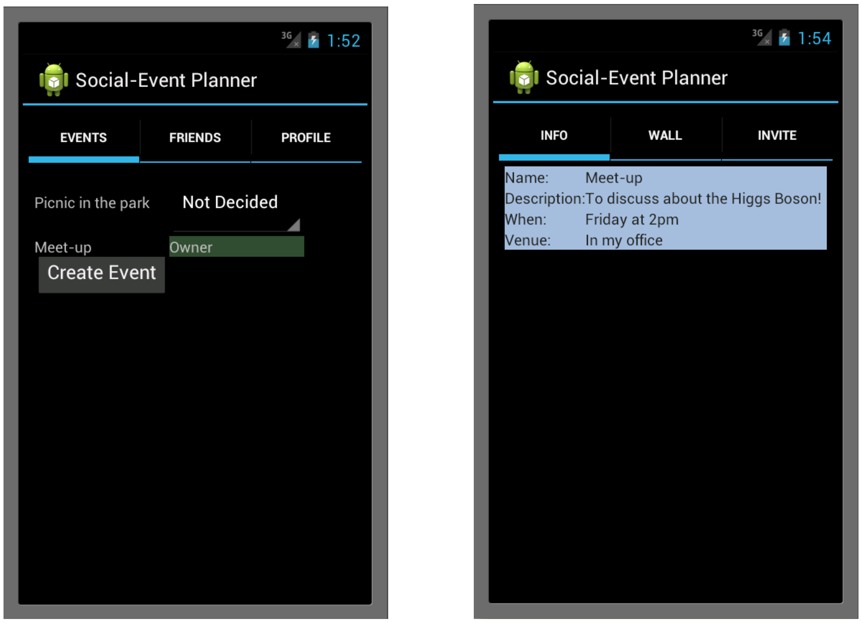
```
public class ref8_socialpermissions{
  public Lock lock = new ReentrantLock(true);
  //@ public static constraint PERSON.equals(\old(PERSON));
  public static final BSet<Integer> PERSON = new Enumerated(min_integer,
       max_integer);

  //@ public static constraint EVENTS.equals(\old(EVENTS));
  public static final BSet<Integer> EVENTS = new Enumerated(min_integer,
       max_integer);
  ...

  /*@ spec_public */ private BSet<Integer> persons;
  /*@ spec_public */ private BSet<Integer> sevents;
  /*@ spec_public */ private BRelation<Integer,Integer> eventowner;
  ...

  /******Invariant definition******/
  /*@ public invariant
    persons.isSubset(PERSON) &&
    sevents.isSubset(EVENTS) &&
    eventowner.domain().equals(sevents) &&
    eventowner.range().isSubset(persons) && eventowner.isaFunction() && BRelation.
         cross(sevents,persons).has(eventowner) &&
    ... */

  // ... getter and mutator method definition

  /*@ public normal_behavior
    requires true;
    assignable \everything;
    ensures
       persons.isEmpty() &&
       sevents.isEmpty() &&
       eventowner.isEmpty() &&
       ... */
  public ref8_socialpermissions(){
    persons = new BSet<Integer>();
    sevents = new BSet<Integer>();
    eventowner = new BRelation<Integer,Integer>();
    ...

    // Thread initialisation
  }
}
```

Figure 7.4: Excerpt of the translation of machine *ref*8*_socialpermissions* to Java

```
public class create_social_event extends Thread{
 /*@ spec_public */ private ref8_socialpermissions machine;

 /*@ public normal_behavior
     requires true; assignable \everything;
     ensures this.machine == m; */
 public create_social_event(ref8_socialpermissions m) {
     this.machine = m;
 }

 /*@ public normal_behavior
     requires true; assignable \nothing;
     ensures \result <==> (machine.get_persons().has(pe)
      && !machine.get_sevents().has(se)); */
 public /*@ pure */ boolean guard_create_social_event(Integer pe, Integer se) {
  return (machine.get_persons().has(pe)
    && !machine.get_sevents().has(se));
 }

 /*@ public normal_behavior
     requires guard_create_social_event(pe,se);
     assignable m.sevents, m.eventowner, ...;
     ensures m.get_sevents().equals(\old((m.get_sevents()
         .union(new BSet<Integer>(se)))))
       && m.get_eventowner().equals(\old((m.get_eventowner()
         .override(new BRelation<Integer,Integer>(
          new Pair<Integer,Integer>(se,pe)))))) && ...;
    also
     requires !guard_create_social_event(pe,se);
     assignable \nothing;
     ensures true; */
 public void run_create_social_event(Integer pe, Integer se) {
  if(guard_create_social_event(pe,se)) {
   BSet<Integer> sevents_tmp = m.get_sevents();
   BRelation<Integer,Integer> eventowner_tmp = m.get_eventowner();
   ...
   m.set_sevents((sevents_tmp.union(new BSet<Integer>(se))));
   m.set_eventowner((eventowner_tmp.override(
          new BRelation<Integer,Integer>(new Pair<Integer,Integer>(se,pe)))));
  ...
  }
 }

 public void run() { ... }
}
```

Figure 7.5: Excerpt of the translation of event *create_social_event* to Java

the Social-Event Planner.



Name: Meet-up
Description: To discuss about the Higgs Boson!
When: Friday at 2pm
Venue: In my office

(a) Screen 1: Part of the user interface for the Social-Event Planner.

(b) Screen 2: Part of the user interface for the Social-Event Planner.

Figure 7.6: Screenshots Social-Event Planner

The user interface is composed of three main screens:

1) where the events (created by the user or invited for someone else) are displayed. The user has the option to reply to a social-event, to create another social-event, or to see the information of a specific social-event (see Figure 7.6a). The information of a specific social-event (see Figure 7.6b) is the name of the event, the description (e.g. date, venue), the list of the invited people (if the user is the owner of the event, there is an option to invite more people) and finally there is a ‘wall’ where the invited people can comment or share content. The user can also reply to an invited event. *2*) The second screen allows the user to see its friends, as well as add/delete more friends. *3*) Finally, the user has the possibility to see/change its personal information.

All the sources and the code generated and implemented for the Social-Event Planner are available at `http://poporo.uma.pt/EventB2Java/EventB2Java_studies.html`. Additionally, Table 7.4 in Section 7.3 (page 145) presents relevant statistics for the Social-Event Planner.

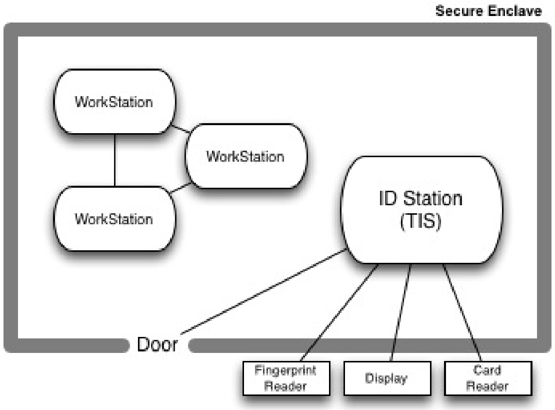


Figure 7.7: The Tokeneer System.

7.2 Tokeneer

The Tokeneer system was developed by Praxis High Integrity. Praxis modelled Tokeneer in Z [6, 102] and implemented it in Spark Ada [10]. The Tokeneer system consists of a secure enclave and a set of system components as shown in Figure 7.7. The Tokeneer ID Station (TIS) is responsible for reading a fingerprint and, based on a number of protocols and checks, ensuring that any person trying to access the enclave is indeed permitted to enter the enclave, and giving the corresponding grants as a user or administrator. The TIS communicates with a number of external components to perform its analysis. The physical devices that are interfaced to the TIS are a fingerprint reader, a smart-card reader, a floppy drive, and a door and a visual display. Individuals enter the secured enclave via the door by providing the credentials either to the fingerprint reader or the card reader. The visual display shows messages that help to track the progress of the user entry process into the secured enclave. An Audit Log logs all events and actions performed or monitored by the TIS. The Token is the card that is inserted by the user to enter the enclave. There are different types of certificates that are used for verification of each Token, and certificates are a crucial part of the Tokeneer system.

The TIS is about 10K lines of code. Praxis wrote the software specifications of the TIS in Z following a System Requirements Specification (SRS) document written by them, and manually translated the Z specification to Spark Ada. The documents described below were written and used by Praxis for developing the TIS and can be found at [47].

- The System Requirements Specification (SRS) includes the TIS software requirements.

- The Formal Specification of the TIS includes the TIS software requirements written in Z.

- The specifications in the above document were later refined and extended in a document called the Formal Design in which operations in Z are extended and more system invariants are considered.
- The System Test Specification (STS) presents the test cases for the TIS.

The following sections describe in more detail the components, and the operations of the TIS. Very detailed information can be found in [47].

7.2.1 TIS Components

TIS is mainly composed of four main physical components used to comunicate the ID Station with the exterior as depicted in Figure 7.7. The TIS contains:

The Door: allows user to enter to the enclave. The door has two possible states, it can be open or closed. It has a latch that can be locked or unlocked, and an alarm.

The Fingerprint Reader: collects information about the fingerprint of the users. It is used to compare if the fingerprint of the user trying to the enclave matches the fingerprint already stored in the system.

The Display: shows short messages to the user on a small display during the attempt to enter to the enclave. For instance a message could be **AUTHENTICATING USER**, **ENTER TOKEN**.

Card Reader: reads the card (token, explained later in this section) that belongs to the user attempting to enter the enclave. The card provides useful information about the user to the system, the system processes this information, and allows (or not) the entering of the user.

7.2.2 TIS Operations

TIS contains a series of operations (dis-) allowing the user of the TIS to perform certain activities. The following presents some concepts necessary to understand the operations:

Certificates: Certificates are used for a user validation during enrolment to the TIS (as explained later in this section). It always contains a unique identifier, and a validity period during which time the certificate is valid. Certificates also have an asymmetric key for verification, that could be optional.

There are different types of certificates in the system. Their hierarchy is shown in Figure 7.8. A Certificate can be an ID Certificate or an Attribute Certificate.

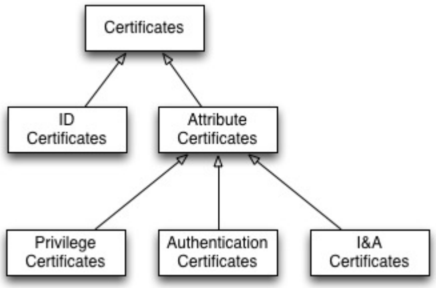


Figure 7.8: Hierarchy of certificate types.

ID Certificate (IDCert) contains a reference to a certificate of the system, the name of the user being identified, and the asymmetric key of the user.

Attribute Certificate (AttCert) contains a reference to a certificate of the system, and a reference of the ID Token related to the certificate. It also contains a reference to an IDCert. An AttCert can be private Certificate (privCert), an Identification and Authentication Certificate (IandACert), or Authorisation Certificate (authCert).

Private Certificate (privCert) contains additional attributes: a role, it can be a user only, a guard a securityOfficer, or an auditManager (role determines privilege over the TIS); and a clearance that determines the ordered classifications on documents, areas, and people. It can be: unmarked, unclassified, restricted, confidential, secret, or top secret.

Identification and Authentication Certificate (iandACert) contains a fingerprint template which contains information reading from a fingerprint, this information is used to compare if the fingerprint of the user being identified matches the information in the system.

Authorisation Certificate (authCert) contains the same structure as privCert. It is used for different checkins to enter the TIS.

Tokens (tokens) are smart cards belong to each user of the system. The smart card contains a unique ID, a series of certificates: idCert, privCert, IandACert, and an optional authCert.

Operations over tokens are

- a token is *valid* if each certificate on it correctly cross-references to the IDCert, and each certificate correctly cross-references to the token ID.

- if the Authorisation Certificate is present, it is *valid* if it correctly cross-references to the token ID, and the IDCert.
- a token is *current* if all certificates on it are current, it means, if the current Time is within the validity period of each certificate.

User Entry Operations

These operations describe the process a user needs to do to be authenticated to further enter the enclave. It is presented as a state transition diagram. Operations are:

User Token Tears: If the user tears the Token out before the operation is complete then the operation is terminated unsuccessfully.

Reading the User Token: This operation performs actions that reads an inserted token.

Validating the User Token: Once TIS has read the user token, the token content needs to be validated. The token passes the validation state if

- the token is *valid* and it contains an authCert Certificate that cross-checks correctly with the token ID and the ID certificate. The token must be *current* and both the authCert and IDCert certificate can be validated. In this case Biometric checking is not performed, or
- the token is *consistent*, *current*, and the IDCert, priviCert, and iandACert can be validated. In this case, Biometric checks will be required, or
- in the case where there is a valid authCert certificate the biometric checks are passed.

The biometric checks are only required if the authCert Certificate is not present or not valid. In this case the remaining certificates on the card must be checked.

Reading and Validating a Fingerprint: During the entering process, users might be asked to provide the fingerprint as a validation process (biometric check). This operation reads users' fingerprint and compares it against the fingerprint information already stored in the system.

Writing the User Token: This operation attempts to write an authorisation certificate in the user's token, it may (un-)successfully written.

Validating Entry and Unlocking Door: The system will validate the entrance of the user, if the user's token passes all checks, the door will be unlocked.

Operations within the Enclave

Users of the TIS may have permission to operate the Enclave. Those users are called administrator and they can performed additional operation with the Enclave. The process to operate the Enclave is also presented as state transition diagram. The following describes these operations

Enrolment of an ID Station: In order for a user to perform administrator operations he has to enrol the TIS. The user needs to request the enrolment to the system providing information in a floppy, the system validates the information, and if the data is valid, enrols the user as an administrator. After this process the user (administrator) needs to provide his token to the system.

Administrator Token Tear: If the administrator tears his token will result in his logging out from the system.

Administrator Log-in: In order for an administrator to log-in the system, the administrator needs to insert a *valid* token into the token reader. If the information provided is valid, the administrator can enter the enclave and will have the privileges indicated in the token.

Administrator Log-out: The logging-out of an administration can happen for either the administrator removes his token from the TIS or the authorisation certificate expires.

Administrator Operations: The administrator has a set of operations to perform. The administrator can archive the log, update the configuration data, overrideLock, or shutdown the system. The privileges administrators have are written in his token.

7.2.3 An example of User Entry Operation

Figure 7.9 is an excerpt of the transition state diagram for users' authentication and entry process. The figure shows just the transition process for a user to enter the enclave with a token that does not contain any authCert certificate, and *valid* and *current* privCert and iandACert Certificates. The system requires the user to pass the biometric checks, and finally writes its token.

In Figure 7.9, the ovals represents the states and the lines represent the operation performed (the transition). The red oval represents the starting and ending point of the diagram. The system starts in a quiescent state. Once the user puts his token in the card reader, the operation `ReadUserToken` is performed. This operation requires the status of the system to be in quiescent, and the token to be present. The operation changes the status to gotUserToken and displays a message in the display ("AUTHENTICATING USER, PLEASE WAIT"). The system evolves and goes to state gotUserToken. Since the token

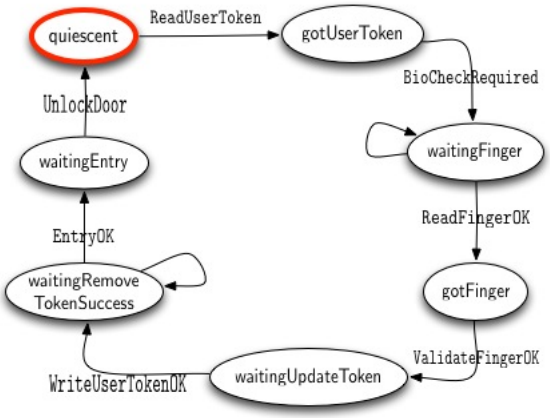


Figure 7.9: Excerpt State Diagram User Entry to TIS.

inserted does not contain authCert Certificate and the privCert and iandACert certificates are *valid* and *current*, the system requires the user for a biometric check. The biometric check is to read the fingerprint. The system evolves to state waitingFinger. The user puts the fingerprint in the finger reader (as stated by transition `ReadFingerOk`). The system goes to state gotFinger. The operation `ValidateFingerOK` checks that the information of the fingerprint read by the fingerprint reader indeed matches with the information stored in the system, if so, the system evolves to waitingUpdateToken that changes the status to `waitingEntry`. Since the status is `waitingEntry`, the token of the user is still inserted, and the certificates on it are *valid* and *current*, the user is granted permission to enter the enclave. The system evolves to the state waitingRemoveTokenSuccess where the system waits until the user remove the token to finally unlock the door.

7.2.4 Conversion from Z to Event-B

This section discusses the strategy we followed to convert the existing model of TIS in Z to Event-B. Z is a notation for writing specifications based on set theory and first order logic. It includes a notation for discrete mathematics (set theory and predicate calculus) and for describing and combining schemas (the schema calculus) that allows one to define possible states as well as operations that can change the state.

In Z, one can define a variable (in capital letters) to express basic types. For instance, the Z schema defined above defines two sets `USER` and `TOKENID` as the sets of all possible users and token ids of the system:

```
[USER]
[TOKENID]
```

That definition goes in the same direction as carrier sets in Event-B. We modelled Z variables in capital letter as carrier sets in Event-B. For instance, the definition of the Z schemas [`USER`] and [`TOKENID`] is in Event-B:

context *ctx*
 sets USERS TOKENID
end

In Z, one can define a schema that defines variables. The schema *Certificate* (shown below) defines a common certificate in Tokeneer: a certificate contains a unique identifier (denoted by variable *id*), it also contains a validity period in which that certificate is valid (denoted by variable *validityPeriod*), and a asymmetric key that validates the certificate (denoted by variable *isValidatedBy*).

Certificate
id : CertificateId
ValidityPeriod : $\mathbb{P}\,\mathtt{TIME}$
isValidatedBy : *optional* `KEY`

In Z, one can instantiate schemas and refer to variables of that schema using the notation dot (.). For instance one can define a certificate *c* and refer the variable *id* as *c.id*. Since this is not possible in Event-B, we decided to model Z schemas in Event-B as several relations that map the type schema to each variable. For instance, the following Event-B machine defines the schema *Certificate* in Event-B:

machine *m* sees *ctx*
variables *certificates certificateID validityPeriods publicKeys isValidatedBy*
invariants
 inv1 $certificates \subseteq \mathsf{CERTIFICATES}$
 inv2 $publicKeys \subseteq \mathsf{KEYS}$
 inv3 $certificateID \in certificates \rightarrowtail \mathsf{CERTIFICATEID}$
 inv4 $validityPeriods \in certificates \mathrel{\twoheadleftarrow\!\!\!\!\leftrightarrow} \mathbb{N}$
 inv5 $isValidatedBy \in certificates \mathrel{\rightarrowtail\!\!\!\!\!\!+} publicKeys$
end

Where context *ctx* defines the carrier sets CERTIFICATES, KEYS, and CERTIFICATEID (not shown here). The model of variables in Event-B follows the Z specification of them and the System Requirement Specification (SRS) document. For instance, just from the Z specification is not possible to deduce that id variable needs to be defined as a total injection. However, reading the SRS document one can realise id variable define a unique id for each certificate in the system. In Z, one can define a variable as *optional*, we model that as a partial injection ($\mathrel{\rightarrowtail\!\!\!\!\!\!+}$ in Event-B), meaning there can be a certificate that does not have any public key associated to it. To create a new certificate one needs to update all variables as:

...
then
act1 $certificates := certificates \cup \{new\_c\}$
act2 $publicKeys := publicKeys \cup \{new\_pk\}$
act3 $certificateID := certificateID \cup \{new\_c \mapsto new\_certid\}$
act4 $validityPeriods := validityPeriods \cup \{new\_c \mapsto new\_valtime\}$
act5 $isValidatedBy := validityPeriods \cup \{new\_c \mapsto new\_pk\}$
end

So to access the identifier value of a certificate c (as in Z is $c.id$), in Event-B one does $certificateID.apply(c)$.

In Z, one can also define a schema composed of two parts divided by a line: above the line one can define variables or import another schemas already defined, below the line one can define predicates. The following Z schema *ReadUserToken* depicts this kind of schema:

[`STATUS` $::= \{quiescent, gotUserToken, waitingFinger\}$]
[`PRESENCE` $= \{absent, present\}$]
[`DISPLAYMESSAGE` $::= \{wait, welcome, insertFinger\}$]

Context
status : `STATUS`
tokenPresence : `PRESENCE`
display : `DISPLAYMESSAGE`

ReadUserToken
Context

status = quiescent
tokenPresence = present
status' = gotUserToken
display' = wait

Z Schema *ReadUserToken* is partially defining one of the states defined in Figure 7.9. The schema is not defining any new variable but it is importing schema *Context* so predicate expressions like $status = quiescent$ can be used. We translate a schema as an Event-B event where the predicate is the guard of the event. Notice that Schema *ReadUserToken* defines a predicate with the aid ($'$), that is how schemas evolve in time. For instance, predicate $status' = gotUsertToken$ means the new value for variable $status$ will be $gotUsertToken$. Z predicates with the aid of $'$ is translated to Event-B in the actions of the event. the following Event-B machine translates the schemas:

```
machine $m$ sees $ctx$
variables $status\ tokenPresence\ display$
invariants
  inv1 $status \in \mathsf{STATUS}$
  inv2 $tokenPresence \in \mathsf{PRESENCE}$
  inv3 $display \in \mathsf{DISPLAYMESSAGE}$

  $ReadUserToken$
   where
    grd1 $status = quiescent$
    grd2 $tokenPresence = present$
   then
    act1 $status := gotUserToken$
    act2 $display := wait$
  end
end
```

Context *ctx* defines carrier sets STATUS, PRESENCE, and DISPLAYMESSAGE, and defines the corresponding axioms. For instance the axiom related to STATUS is

$$\textbf{ax1}\ \mathsf{STATUS}\ = \{quiescent, gotUserToken, waitingFinger\}$$

Notice that the Event-B translation of the Z schema *Context* does not define variables as relation (as shown for Z schema *Certificate*). Our decision on doing that is that schema *Context* defines variables *status*, *tokenPresence*, and *display* that are always the same during the execution of the system, whereas schema *Certificate* can be instantiated any time a new certificate is created.

We follow the notion of translation described in this section to translate the TIS system modelled in Z to Event-B.

7.2.5 Modelling the Tokeneer ID Station (TIS) in Event-B

We modelled the Tokeneer ID Station (TIS) in Event-B following the Z model of the TIS and the documentation provided by Praxis. We followed the "parachute" software development strategy of Event-B proposed by J.-R Abrial in [2]. Table 7.1 lists a few software requirements of the TIS. The table includes some functional (`FUN`) and environmental (`ENV`) requirements. We wrote an abstract machine, six machine refinements and an additional AuditLog machine as shown in Table 7.2.

The abstract machine models certificates. The first and the second refinements include specialised certificates. These first three machines model certificates and its hierarchy as shown in Figure 7.8. The third refinement models fingerprints and the internal status to enter the enclave. The fourth refinement models entry to the enclave and the display used by the TIS. The fifth refinement models enrolments to the enclave using a certificate, and the sixth refinement models some administrative functionality. Machine *AuditLog* models a log of all events and actions performed or monitored by the TIS.

Req.	Description
`FUN-1`	Certificates have a unique ID, a period during which they are valid, and a public key of the user (used to sign and verify the certificate).
`FUN-2`	The system contains two kinds of certificates: ID and Attribute certificates. They are used during enrolment and are present on tokens.
`FUN-3`	Attribute certificates are categorised into three types as follows: Authorisation and Privilege certificates which have the same structure, and I&A certificates.
`FUN-4`	Tokens are used to store all the required information of a user. Each token contains ID, Privilege, I&A, and Authorisation (optional) certificates.
`ENV-1`	Tokens are the data read from an inserted Smart Card. The system contains a card reader to read tokens.
`FUN-5`	Tokens should be *valid* (certificates correctly cross reference to the ID Certificate) and *current* (all included certificates are up to date) before processing.
`FUN-6`	TIS enrolment transition is defined by the transition state diagram in [47] (41_2.pdf/pp. 59/ Fig. 7.1).
`ENV-2`	The system contains a floppy drive
`FUN-7`	The TIS maintains an audit log with fixed size that logs all the actions taken within the enclave.
`FUN-8`	TIS administrators are users with higher security privileges. Administrators may log on to the TIS console, log-off, or start an operation.
`ENV-3`	The System contains a fingerprint reader. Fingerprints are used if bio-metric security is required.
`ENV-4`	The System contains a small display outside the enclave.
`ENV-5`	The door controls user entry to the enclave, and is either open or closed.
`FUN-9`	The user entry transition is defined by the transition state diagram in [47] (41_2.pdf/pp.43/ Fig.6.1), which tracks progress through user entry.

Table 7.1: System Requirements Specification of TIS

Machines	**Level**	**Description**
Abstract	`FUN-1`	Basic certificates (Level-0).
Ref-1	`FUN-2`	IDCert, AttCert (Level-1)
Ref-2	`FUN-3`	PrivCert, AuthCert, I&ACert (Level-2)
Ref-3	`ENV-1`, `FUN-4`, `FUN-5`	Token, fingerprint, and internal status to enter the enclave
Ref-4	`ENV-4`, `ENV-5`, `FUN-9`	Entry to the enclave, the display
Ref-5	`FUN-6`	Enrolment
Ref-6	`FUN-8`	Admin
AuditLog	`FUN-7`, `ENV-2`	Audit Log

Table 7.2: Refinement Strategy for the Tokeneer system.

Figure 7.10 shows an excerpt of the third refinement of the TIS Event-B model. Machine *ref3_entry_L1* models the entry of an user to the enclave. The machine sees context *ctx_ref3* (not shown in the figure) that defines carrier sets CERTIFICATES (the set of all type of certificates), KEYS (the set of asymmetric keys used for signing and validating certificates), TOKENID (the set of all tokens), and ENTRY_STATUS (the set of all possible status for entering the enclave). Variables *certificates*, *publicKeys*, *isValidatedBy*, and *validityPeriods* define the properties of each certificate as specified by the functional requirement FUN-1. Variable *attCert* defines the properties of one kind of certificate (Attribute Certificates) as specified by the functional requirement FUN-2. Variables *privCert*, *iandaCert*, and *authCert* define the properties of Attribute Certificates as specified by the functional requirement FUN-3. Variables *tokenPrivCert*, *tokenIandaCert*, *tokenAuthCert*, *tokenID*, and *attCertTokID* define the properties for tokens as specified by the functional requirement FUN-4. Variable *entry_status* defines the status on the entry of a user to the enclave. For instance, the diagram depicted in Figure 7.8 shows an excerpt of the process of a user to enter the enclave. The states of the diagram are represented by variable *entry_status*, it starts with the value quiescent, after having checked the user token, the system goes to next state, giving the value of gotUserToken to *entry_status*. Variable *currentToken* models the token is being read from the card reader.

Figure 7.10 shows a single event of machine *ref3_entry_L1*, event *BioCheckRequired*. Once the user that wants to enter to the enclave puts its token into the card reader, the system reads the information within the token smart card. A biometric check is required when

```
machine $ref3\_entry\_L1$ sees $ctx\_ref3$
variables $certificates\ publicKeys\ isValidatedBy\ validityPeriods\ attCert$
  $privCert\ iandaCert\ authCert\ tokenPrivCert\ tokenIandaCert$
  $tokenAuthCert\ entry\_status\ currentToken\ tokenID\ attCertTokID$
invariants
  inv1 $certificates \subseteq \mathsf{CERTIFICATES}$  inv2 $publicKeys \subseteq \mathsf{KEYS}$
  inv3 $validityPeriods \in certificates \mathrel{\twoheadleftarrow\!\!\!\!\rightarrow} \mathbb{N}$
  inv4 $isValidatedBy \in certificates \mathrel{\rightarrowtail\!\!\!\!\!\!+}\ publicKeys$
  inv5 $attCert \subseteq certificates$  inv11 $entry\_status \in \mathsf{ENTRY\_STATUS}$
  inv6 $\mathsf{partition}(attCert, privCert, iandaCert, authCert)$
  inv7 $tokenID \subseteq \mathsf{TOKENID}$  inv12 $currentToken \in \mathsf{TOKENID}$
  inv8 $tokenPrivCert \in tokenID \rightarrowtail privCert$
  inv9 $tokenIandaCert \in tokenID \rightarrowtail iandaCert$
  inv10 $tokenAuthCert \in tokenID \mathrel{\rightarrowtail\!\!\!\!\!\!+}\ authCert$
  inv13 $attCertTokID \in attCert \rightarrowtail tokenID$
 $BioCheckRequired$
  any $currentTime$ where
   grd1 $entry\_status = gotUserToken$  grd2 $currentTime \in \mathbb{N}$
   grd3 $(currentToken \in dom(tokenAuthCert) \wedge$
         $currentTime \notin validityPeriods[tokenAuthCert(currentToken)])$
         $\vee currentToken \notin dom(tokenAuthCert)$
   grd4 $currentToken \in ran(attCertTokID) \wedge$
         $attCertTokID(currentToken) \in dom(isValidatedBy)$
   grd5 $currentToken \in dom(tokenPrivCert) \wedge$
         $tokenPrivCert(currentToken) \in dom(isValidatedBy)$
   grd6 $currentToken \in dom(tokenIandaCert) \wedge$
         $tokenIandaCert(currentToken) \in dom(isValidatedBy)$
  then
   act1 $entry\_status := waitingFinger$
 end
end
```

Figure 7.10: Excerpt third refinement machine TIS Event-B model

- the status of the entry is $gotUserToken$ as stated by guard **grd1**,
- the user token is valid for entry into the enclave, i.e if the token
  - is consistent (e.g. $currentToken \in dom(tokenPrivCert)$),
  - ID certificate, Privilege certificate and IandA certificate can be validated (e.g. $tokenPrivCert(currentToken) \in dom(isValidatedBy)$) as stated by guards **grd4**, **grd5**, and **grd6**, and
- the Authorisation Certificate is not present (e.g. $currentToken \notin dom(tokenAuthCert)$) or not valid (e.g. $currentTime \notin validityPeriods[tokenAuthCert(currentToken)]$) as stated by guard **grd3**.

```
public class ref3_entry_L1{
  BioCheckRequired evt_BioCheckRequired = new BioCheckRequired(this);

 public static final BSet<Integer> CERTIFICATES = new Enumerated(INT.min,INT.max);
  // ...  definition of the rest of carrier sets

  private BSet<Integer> attCert;
  private BRelation<Integer,Integer> isValidatedBy;
  // ...  definition of the rest of variables

  // ... definition of getter and mutator methods

  public ref3_entry_L1(){
    attCert = new BSet<Integer>();
    isValidatedBy = new BRelation<Integer,Integer>();
    // ... initialisation of class fields

    }
}
```

Figure 7.11: Partial translation of machine $ref3\_entry\_L1$

The biometric check consists in reading the fingerprint of the user so the system can compare it against the fingerprint already stored in the system. If a biometric check is required, the system goes to state waitingFinger as stated by the action **act1**.

7.2.6 Generating Java code for the TIS Event-B model

After modelling TIS in Event-B and discharging all proof obligations, we generated Java code of the model using EventB2Java. Figure 7.11 depicts an excerpt of the translation of the machine $ref\_3\_entry\_L1$ and Figure 7.12 shows an excerpt of the translation of the event *BioCheckRequired*.

Figure 7.11 defines carrier sets, variables, and a constructor of the class.

Figure 7.12 shows a partial translation of event *BioCheckRequired* where `machine` is a reference to the machine class implementation. The Java code includes methods `guard_BioCheckRequired` (the translation of the event guard) and `run_BioCheckRequired` (the translation of the event body). The JML specifications generated by EventB2Java are omitted since the specifications are not used in generating tests or customising code in this example.

7.2.7 Writing JUnit Tests

Software Testing [17] can be used to validate software requirements that are expressed in a formal language. A common way of testing is the formulation of expected results. Hence, testing is achieved by comparing the results from executing the system against the expected ones.

The System Test Specification of the TIS includes 32 test cases organised in eight categories as shown in Table 7.3. We wrote Java code for these 32 test cases in two steps. We first used the EventB2Java tool to translate the Event-B model of the TIS to Java. We generated a sequential version of the model in

```
public class BioCheckRequired{
  private ref3_entry_L1 machine;

  public BioCheckRequired(ref3_entry_L1 m) {
    this.machine = m;
  }

  public boolean guard_BioCheckRequired(Integer currentTime) {
    return (
        machine.get_entry_status().equals(machine.gotUserToken) &&
        machine.get_tokenAuthCert().domain().has(machine.get_currentToken()) &&
        !machine.get_validityPeriods().image(new BSet<Integer>(machine.
             get_tokenAuthCert().apply(machine.get_currentToken()))).has(
             currentTime) || !machine.get_tokenAuthCert().domain().has(machine.
             get_currentToken()) && ...;
    }

  public void run_BioCheckRequired(Integer currentTime){
    if(guard_BioCheckRequired(currentTime)) {
      Integer entry_status_tmp = machine.get_entry_status();
      machine.set_entry_status(machine.waitingFinger);
    }
  }
}
```

Figure 7.12: Partial translation of event *BioCheckRequired*

Java since the tests are run sequentially. We then gave initial values for Java constants that respect the axioms on those constants defined in the Event-B model.

We ran the 32 JUnit tests using the input data provided by Praxis. Then we compared the obtained results against the expected results also provided by Praxis. As an example of a test, Praxis defined *UserEntry1* as one of the test cases of the `UserEntry` category. The test allows an administrator with role "Security Officer" to enter the enclave and acquire a valid Auth Certificate. The test follows the state diagram presented in Figure 7.9, it goes through the following steps:

- ReadUserToken
- BioCheckRequired
- ReadFingerOK
- ValidateFingerOK
- ConstructAuthCert
- WriteUserTokenOK
- EntryOK
- UnlockDoorOK

Category Test	**Description**
`Enrolment`	3 tests for starting an un-enrolled TIS attempting to enroll using different types of certificates.
`UserEntry`	14 tests for allowing administrators with different roles and users with different kinds of certificates to enter the enclave.
`UpdateConfig`	5 tests for allowing different kinds of administrators to update the configuration of the enclave.
`Override`	1 test for overriding the operation of the door by the guard administrator.
`Admin Login`	3 tests for allowing an administrator to log-in to the TIS.
`Admin Logout`	1 test for logging-out of the TIS.
`Shutdown`	2 tests for shutting down the system.
`ArchiveLog`	3 tests for actions over the AuditLog component.

Table 7.3: System Test Specification of the TIS.

Figure 7.13 shows the JUnit implementation of test *UserEntry1*. Variable `machine` (a reference to the machine in the Java implementation) gives access to all the variables and events of the model. Method `set_test_UserEntry1` is used to initialise variables, variables are initialised according to initial values given by Praxis. When executed, this test will fail if any `guard_evt` method returns **`false`**, or if any `run_evt` method does not set the proper screen and display messages, as stated by Praxis documentation in Expected Results. The final result of this test matches the expected result: the messages on the screen were correct, and Authorisation Certificate was created, and the door is open so user can enter to the enclave.

During the first round of testing, the Java code did not pass all 32 JUnit tests. We inspected the Event-B model and discovered that the model was creating a specialised Authorisation Certificate for a user in the wrong event. As this error did not invalidate the model, it could not be detected via model verification in Event-B. We corrected the Event-B model, discharged all the proof obligations again, and used the EventB2Java tool to regenerate the Java code. We repeated this process until the code passed all 32 JUnit tests. Our Event-B model of the TIS, the Java code generated by the EventB2Java tool, and the 32 JUnit tests that we wrote can be found at `http://poporo.uma.pt/Tokeneer.html`.

There are several benefits that can be obtained by applying our strategy for testing. The user gains confidence in the correctness and appropriateness of the modelled system by discharging all the Event-B proof obligations in Rodin. The JUnit tests provide an additional layer of confidence by checking that the

```
    @Test
    public void test_UserEntry1(){
    set_test_UserEntry1(machine);

    // ReadUserToken
    Assert.assertTrue("Guard evt_ReadUserToken not satisfied.", machine.
         evt_ReadUserToken.guard_ReadUserToken(token_user_to_read));
    machine.evt_ReadUserToken.run_ReadUserToken(token_user_to_read);
    Assert.assertEquals(machine.get_displayMessage1(), ref6_admin.wait);

    // BioCheckRequired
    Assert.assertTrue("Guard evt_BioCheckRequired not satisfied.", machine.
         evt_BioCheckRequired.guard_BioCheckRequired(currentTime));
    machine.evt_BioCheckRequired.run_BioCheckRequired(currentTime);
    Assert.assertEquals(machine.get_displayMessage1(), ref6_admin.waitingFinger);

    machine.set_FingerPresence(machine.present);
    // ReadFingerOK
    Assert.assertTrue("Guard evt_ReadFingerOK not satisfied.", machine.
         evt_ReadFingerOK.guard_ReadFingerOK());
    machine.evt_ReadFingerOK.run_ReadFingerOK();
    Assert.assertEquals(machine.get_displayMessage1(), ref6_admin.wait);

    Integer fingerPrint = User01fp;
    // ValidateFingerOK
    Assert.assertTrue("Guard evt_ValidateFingerOK not satisfied.", machine.
         evt_ValidateFingerOK.guard_ValidateFingerOK(fingerPrint));
    machine.evt_ValidateFingerOK.run_ValidateFingerOK(fingerPrint);
    Assert.assertEquals(machine.get_displayMessage1(), ref6_admin.wait);

    // ConstructAuthCert -> built-in writeUserTokenOK
    // WriteUserTokenOK
    Assert.assertTrue("Guard evt_WriteUserTokenOK not satisfied.", machine.
         evt_WriteUserTokenOK.guard_WriteUserTokenOK(cert_params));
    machine.evt_WriteUserTokenOK.run_WriteUserTokenOK(p_id_cert, p_priv, p_ce,
         p_tid, p_serial, p_issuer, p_period, p_pubkey, p_class);
    Assert.assertEquals(machine.get_displayMessage1(), ref6_admin.wait);

    // EntryOK
    Assert.assertTrue("Guard evt_EntryOK not satisfied.", machine.evt_EntryOK.
         guard_EntryOK(currentTime));
    machine.evt_EntryOK.run_EntryOK(currentTime);
    Assert.assertEquals(machine.get_displayMessage1(), ref6_admin.openDoor);

    machine.set_userTokenPresence(machine.absent);
    // UnlockDoorOK
    Assert.assertTrue("Guard evt_UnlockDoorOK not satisfied.", machine.
         evt_unlockDoorOK.guard_unlockDoorOK(currentTime));
    machine.evt_unlockDoorOK.run_unlockDoorOK(currentTime);
    Assert.assertEquals(machine.get_displayMessage1(), ref6_admin.doorUnlocked);
}
```

Figure 7.13: The *UserEntry1* Test Case in JUnit

Event-B Model	**LOC**	**# Mch**	**# Evt**
Social-Event Planner [90]	1326	9	35
MIO [37]	586	7	21
Heating Controller [56]	458	15	32
State Machine [97]	86	2	5
Binary Search [3]	101	3	3
Linear Search [3]	54	2	2
Minimum Element [3]	64	2	3
Reversing Array [3]	64	2	2
Sorting Array [3]	137	3	4

Table 7.4: Statistics of the Event-B Models

behaviour of the Event-B model in Java is what the user actually intended. The Java code generated by the EventB2Java tool is an actual initial implementation of the Event-B model that can be used as is, or further refined and customised as needed.

7.3 Comparing EventB2Java

We are interested in comparing our EventB2Java tool against other tools that generate Java implementations from Event-B models. In particular, we have compared EventB2Java with Code Generation [54, 55] by A. Edmunds and M. Butler, and EB2J [77] by D. Méry and N. Singh. Although Code Generation can generate Ada code in addition to Java, we were interested in examining and analysing its ability to generate Java code only. Likewise, EB2J is able to generate C, C++ and C# code, but we did not consider this in our comparison.

The comparison defines a set of six performance criteria as follows. *i*) "Generation Process" – does the user need to adapt the Event-B model before using the tool to generate Java code. It might be *a*) "Automatic", if the user does not need to edit or extend the Event-B model, or *b*) "Assisted", if the user does need to do so, or *c*) "Automatic/Assisted", if the user needs to do so in some cases and does not in other. *ii*) "Executable" – does the generated code compile and run as is. *iii*) "Support for Code Customisation" – does the tool furnish a mechanism for the user to be able to customise the generated code and to verify whether the customised code is correct. *iv*) "Support for Event-B's Syntax" – does the tool *a*) "Fully", *b*) "Largely", or *c*) "Scarcely" support the current syntax of Event-B. *v*) "Execution Time" – how long does it take for the generated code to execute and to give a result (if the execution terminates). Finally, *vi*) "Effective Lines of Code" – the actual number of lines of Java code generated by the tool.

In addition to defining a set of performance criteria, we need to provide a fair context for comparing tools. We selected the nine Event-B models shown in Table 7.4. We developed two of the systems – the Social-Event Planner

Tool	**Gen. Proc.**	**Exec. Code**	**EB support**	**Code Custom.**
EventB2Java	Aut./ Ast.	Yes	Largely	Yes
Code Gen. [54]	Ast.	Yes	Fairly	No
EB2J [77]	Ast.	Yes	Scarcely	No

Table 7.5: Tool Comparison

[90] and the MIO model [37]. The Social-Event Planner is presented as a case study in Section 7.1. MIO is an Event-B model of a massive transportation system that includes articulated buses following the main corridor routes of a city (briefly described in Chapter 4). The Heating Controller [56] and the State Machine [97] models were developed by one of our tool competitors. The Heating Controller is an Event-B model of a heating controller that provides an interface to adjust and display a target temperature, and to sense and display the current temperature, among other functionality. State Machine is an Event-B model of state machines. The rest of the examples in Table 7.4 are sequential program developments written by J.-R. Abrial in [3]. Linear and Binary Search are the Event-B models of the respective searching algorithms. Minimum Element is an Event-B model for finding the minimum element of an array of integers. Reversing and Sorting Array are Event-B models for reversing and sorting an array respectively. Square Root Number is an Event-B model for calculating the square root of a number.

Table 7.4 presents some statistics about the Event-B models used in the comparison. "LOC" stands for Lines of Code in Event-B, and "# Machines" and "# Events" for the number of machines and events respectively of the Event-B model. EventB2Java successfully generated JML-annotated Java code for all the models in Table 7.4 – we were able to run the Java code as generated in each case. All of the examples in Table 7.4 are available from `http://poporo.uma.pt/EventB2Java/EventB2Java_studies.html`. The site includes the Event-B models and the Eclipse projects with the generated JML-annotated Java implementations. The Eclipse projects also include test files that can be used to run the Java code. These test files are generated automatically by EventB2Java, except in cases where the Event-B models make use of axioms. In those cases, we wrote and added the test files manually. For example, Binary Search defines a constant v to be the searched for value, and a function f to be the array containing the values, so that $v \in ran(f)$. For the EventB2Java generated code to work, one needs to manually assign a value to v that is in the array f. In writing a file to test the Java code of the of Binary Search algorithm, one must consider those conditions on v and f.

Table 7.5 shows how the tools considered in our comparison compare on the criteria of Generation Process (**Gen. Proc.**), Executable Code (**Exec. Code**), Support for Event-B's Syntax (**EB Support**) and Support for Code Customisation (**Code Custom.**). Regarding "Generation Process", EB2J and Code Generation are (always) "Assisted" (Ast.) since tool users (always) need to modify (extend) the Event-B model for the tools to be able to generate code. EventB2Java is "Automatic/Assisted" (Aut/Ast.). More precisely, it is "Automatic" in all cases except when the Event-B model makes use of axioms. As EventB2Java does not yet generate Java code for axioms (which constrain the values of constants), the user must choose values for those constants. EventB2-Java does generate JML specifications for axioms, so the user can employ JML machinery [25] to confirm that the values chosen are valid with respect to the original Event-B model.

The Code Generation tool (Code Gen.) is "Assisted" as it always requires the user to employ the Event Model Decomposition Rodin plug-in [5] to decompose Event-B models into sub-models. For example, if the Event-B machine models the system and the environment components of a reactive system, then the plug-in can generate each part separately. In addition to decomposing the model, users of Code Generation have to explicitly specify the execution order for events in the Java implementation. If the Event-B model includes axioms and constants, tool users need to conjecture values for the constants in Event-B and use the Rodin platform to discharge related proof obligations.

Regarding the comparison criterion "Support for Event-B's Syntax" (**EB. Support**), EventB2Java largely supports Event-B's syntax, in part by generating and using libraries supporting Event-B syntax in Java as described in Section 5.3.2. None of the three tools in the comparison can translate non-deterministic assignments to Java (although EventB2Java does generate JML specifications for them). EB2J and Code Generation require the user to write a final Event-B refinement that does not include non-deterministic assignments. The EB2J tool "Scarcely" provides support for Event-B's syntax and so users are required to furnish an additional Event-B refinement that only uses the syntax supported by the tool. For instance, EB2J is unable to translate the invariant **inv** $pages \in contents \mathrel{\twoheadleftarrow\!\!\!\!\twoheadrightarrow} persons$ that states that $pages$ is a total surjective relation that maps $contents$ to $persons$. For EB2J to support the syntax of that invariant, the user has to write an Event-B model refinement that includes the definition of a total surjective relation, e.g. through the three invariants shown below.

invA $owner \in contents \leftrightarrow persons$
invB $dom(owner) = contents$
invC $ran(owner) = persons$

Table 7.5 indicates whether the code generated by each tool is executable as generated. However, there were cases in which EB2J was incorrect. For example, for the Minimum Element model, the tool was unable to infer the type of the constant n, which is defined as natural number greater than 0 and represents the number of elements in the array to be searched. EB2J issued the message "/* No translatable type found for [n] */". EB2J was also unable to infer the types of constants n, f, and variable g in the Reversing Array example. f is the array to be reversed, defined as a function mapping from $1 \mathrel{..} n$ to the set of integers, and g is the reversed array. Finally, EB2J did not translate parallel assignments

	Code Gen.	EB2J	EventB2Java
Social Event Planner	N/A	N/A	1531 (+391)
MIO	N/A	N/A	825 (+272)
Heating Controller	285	N/A	1612 (+418)
State Machine	48	N/A	198 (+62)
Binary Search	N/A	N/A	71 (+33)
Linear Search	N/A	N/A	48 (+31)
Minimum Element	N/A	68	68 (+46)
Reversing Array	N/A	66	55 (+39)
Sorting Array	N/A	79	92 (+64)
Square Root Number	60	51	53 (+31)

Table 7.6: eLOC for the generated Code

properly for the Reversing, Sorting Array, and the Square Root Number models. For example, EB2J translated $g := g \mathbin{\lhd\!\!-} \{i \mapsto g(j)\} \mathbin{\lhd\!\!-} \{j \mapsto g(i)\}$ as `g[i] = g[j]; g[j] = g[i]`. However, this translation is incorrect since assignments in Event-B are to be executed simultaneously.

EventB2Java is the only tool that provides support for "Code Customisation". The JML specifications generated by EventB2Java enable users to replace (parts of) the code generated by EventB2Java with bespoke implementations. Thus, the user may customise the generated implementation and then use JML machinery [25] to verify the customised implementation against the JML specification generated by the EventB2Java tool.

Table 7.6 shows the eLOC (Effective Lines of Code) generated by each tool. eLOC is a measure of all logical lines in the Java code, and does not include blank spaces, comments, specifications, or single curly brackets. We used the ELocEngine software [50] to calculate eLOCs. As shown in the table, the Code Generation tool was able to generate Java code for only three of the ten Event-B models. We were unable to decompose the remaining seven models (marked as "N/A") since they included many variables, which made it too challenging. The EB2J tool was able to generate code for four out of the ten Event-B models. However, the generated code contained minor errors in Java that we were able to fix. The errors concerned inferring the types of some variables and translating parallel assignments as explained above. For the remaining six models, EB2J issued only one error message. The Binary Search model uses universal quantification, which is not supported by the tool. EventB2Java was able to generate JML-annotated Java code for all models, and this code compiled and ran in each case. In particular, the universally quantified assertion mentioned above appeared in an axiom, which EventB2Java translates to JML but not Java. In Table 7.6, the number in parentheses for EventB2Java gives the number of lines of JML specifications generated for each model.

Finally, the Event-B models for Binary and Linear Search, Minimum Element, and Reversing and Sorting Arrays include events whose guards are mutually exclusive. Hence, we used EventB2Java and EB2J to generate (sequential)

Array Size	**Sorting Array**		**Reverse Array**		**Minimum Array**	
	EventB2-Java	EB2J	EventB2-Java	EB2J	EventB2-Java	EB2J
100,000	23	13093	264	1	29	0
200,000	28	51910	258	55	28	1
300,000	37	182311	198	305	30	1
400,000	152	329614	416	406	32	1
500,000	172	497133	457	548	28	1

Table 7.7: Execution times in milliseconds for the Java code generated by EventB2Java and EB2J for the Sorting, Reverse and Minimum Array Event-B models.

Java implementations for each of these models, and because the generated implementations always complete execution, compared the times the generated implementations took to complete for various inputs. In each case, we ran the implementations 10 times and took the average time. Table 7.7 shows how the times compare for the Sorting, Reverse Array and Minimum Array models. For the Sorting Array model, the code generated by EventB2Java outperformed that generated by EB2J. For the Minimum Array model, EB2J outperformed EventB2Java, though times are close. For the Reverse Array model, EB2J outperformed EventB2Java as well, although EventB2Java approaches EB2J as the input size gets larger. The experiment shows that both tools generate runnable implementations for the considered Event-B models. For EventB2Java, the Java classes that implement the Event-B mathematical constructs exhibit good performance, especially when dealing with large inputs. This is due to the implementation using the `TreeSet` Java class. EB2J did outperform EventB2Java in some cases. We believe that this is largely due to the implementation of method `apply` (applying a relation to a set of elements) of class `BRelation`. In EventB2Java, the method `apply` iterates over each element of the relation, so searching for an element is $O(n)$ and searching for k elements is $O(k * n)$. EB2J uses arrays to store relations, so applying a relation to a set is linear in k.

All times reported in Table 7.7 were collected by running the Java code generated by EventB2Java and EB2J on a Mac OS X laptop with an Intel Core i5 2.3 GHz processor. The Event-B models, generated code and timing harness used are available at `http://poporo.uma.pt/EventB2Java/tests.zip`.

7.4 Conclusion

In this chapter we presented two case studies on software development using EventB2Java, demonstrating the effectiveness of using the EventB2Java tool: the first case study was the implementation of a Social-Event Planner Android application developed using a Model-View-Controller (MVC) design pattern; the second case study was the testing of Tokeneer, a security-critical access control system. We also presented a benchmark comparing EventB2Java against two existing tools for generating Java code from Event-B models. The benchmark was composed of the 9 Event-B models and 6 comparison criteria.

Our experience on developing the first case study suggests that software developers can benefit of EventB2Java in several ways: modelling in Event-B enables users to define properties that the software needs to preserve. For instance, regarding permissions over social-events, an interesting property is that invited people to a social-event have permissions over that event to view and edit its contents. We formalised this property (and many others) as Event-B invariants. We proved that the model was consistent by discharging all proof obligations. We were sure that the Java code generated by EventB2Java respects those properties; software developers can also benefit of EventB2Java since they do not have to refine the Event-B model until it is close to a machine implementation whereas the software developers decide that the model has enough details to be translated to Java using EventB2Java; finally, software developers can benefit of EventB2Java since having a Java implementation of an Event-B model allows the model to interact with other implementations. For instance, the Java code generated by the tool represents the Model in a MVC development that can interact with the implementation of the Controller in Java, and the implementation of the View in Android.

Our experience on developing the second case study shows us that software developers can benefit of EventB2Java since the tool can be used in testing the correct behaviour of an Event-B model by translating it to Java and performing JUnit tests in Java. The process allows system developers to be sure that the behaviour of the model is indeed the behaviour that they intended from the beginning. The process of developing the second case study, initially shows us that, even though the Event-B model was correct (all proof obligations were discharged), the model was not behaving according to our intentions. The JUnit tests uncovered an issue in the Java code generated by the tool for one of the Event-B events. We inspected that event in Event-B, found and corrected the error. We discharged again the proof obligations to be sure the model was still consistent, and used EventB2Java to generate again Java code of the model. We repeated this process until the generated code passed all 32 JUnit test cases. The final generated code is an actual implementation of a model in Event-B that was proven correct, and the code is behaving according to what we expect.

The benchmark showed us that EventB2Java outperforms other Java code generators for Event-B in several ways: EventB2Java generates (and embeds) JML specifications in the Java code. That enables users to customise the Java code to further check if the customised code does not invalidate the initial model. We found out that the EB2J and Code Generation tools do not support code customisation; the generation of Java code process in EventB2Java is automatically/assisted whilst for the other two tools is always assisted. This makes our tool more useful since it is easy to use.

We are planning on undertaking a more complex case study where experts in modelling in Event-B and experts in developing in Java (and JML) can work together.

Chapter 8

Future Work

The work presented in this thesis can be extended in different ways. Figure 8.1 depicts my future work (in red), which is explained below. The figure shows (in black) the work done during this thesis.

- Currently, proof obligations generated by EventB2Dafny are manually fed into Dafny. We are planning on integrating EventB2Dafny to Dafny and Microsoft Visual Studio, so the process of discharging POs is automatic and Rodin can directly have feed back from Dafny (this is depicted in Figure 8.1 as (1)). We also want to investigate and to characterise the type of POs for which Boogie outperforms existing Rodin proof-engines.

- Rodin provides a *lasso* functionality whereby users can select or deselect hypotheses of a proof obligation having common variables with the variables of the goal. We plan to implement a similar functionality for EventB2Dafny.

- In [39], Cataño et. al. proposed a proof of soundness of the translation from Event-B models to JML specifications. The proof takes into account any Event-B substitutions, invariants, and the standard Event-B *initialising* event. We are planning on extending the proof to fully prove the translation of Event-B machines and contexts to JML, and proving the soundness of the translation from Event-B to Java code (this is depicted in Figure 8.1 as (2)). Providing a proof of soundness of our EventB2Java tool gives the user confidence about the generated JML-annotated Java code.

- One major frustration in our work is the inadequate tool support for verifying Java programs with respect to JML specifications. Existing verification tools such as KeY [64] and Krakatoa [75] can not handle the full syntax of Java and JML, particularly with regard to generics. We would like to undertake a case study on replacing parts of the code generated by EventB2Java with bespoke implementations and then verifying those

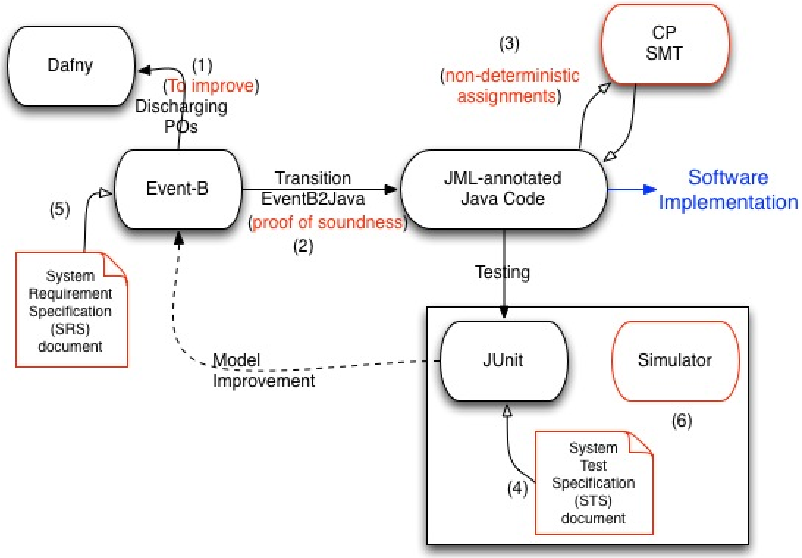


Figure 8.1: Future Work.

implementations against the generated JML specifications. However, performing such verification without adequate tool support is time consuming and prone to error.

- EventB2Java translates Event-B axioms as JML **`static invariants`**. These axioms determine the possible values that constants can take. EventB2Java cannot automatically generate values for constants that satisfy the axioms. We plan on investigate translating the Event-B definitions of constants and axioms to the input language of the Z3 SMT solver [81], and then using Z3 to find values for the constants. Another solution could be to propose a constraint system where constants in Event-B are represented as variables in Java, and Event-B axioms and theorems as constraints, and then use a Java constraint library like CHOCO [44] to obtain correct values for these variables (this is depicted in Figure 8.1 as (3)).

- In Section 7.2 we showed a case study where EventB2Java was applied for testing the behaviour of an Event-B model by translating it to Java using EventB2Java and perform manually written JUnit test. An issue with this strategy is users can introduce errors while manually writing the JUnit tests. We are planning on automating this process so to avoid error prone. In [32], a strategy called JFly is proposed to evolve informal (written in natural language) software requirements into formal requirements written in JML. This work can be reused to structure the writing of JUnit tests from a System Test Specification document (this is depicted in Figure 8.1 as (4)). Likewise, we are planning on investigating a way to map Event-B requirements with a System Requirements Specification document to avoid error prone. In [68], an approach is proposed to obtain a correct representation of requirement specifications in Event-B from a specific semantics also proposed in that approach (this is depicted in Figure 8.1 as (5)).

- As a validation step of an Event-B model behaviour, we proposed to translate the model to Java and perform JUnit tests (as stated in the previous bullet). Another validation step is to simulate the Event-B model so user can see if its behaviour matches the user expectations. In [104], a JavaScript simulation framework for Event-B (JeB) is proposed. It translates Event-B models to JavaScript in order to animate the Event-B model. The idea goes in the same direction as ProB [72]. We are planning on extending the EventB2Java tool so it can generate this kind of simulation so users can check the behaviour of the Event-B model (this is depicted in Figure 8.1 as (6)). Our proposal will outperform JeB in that EventB2-Java generates Java code that serves as a final implementation. The idea is also to have a unique framework that comprises everything: users can obtain JML-annotated Java code from Event-B models, they can check if the behaviour of the Event-B model is the expected by performing JUnit tests or simulating the model in Java.

- Although the modelling of timing properties is not directly supported by Event-B, a discrete clock can certainly be designed and implemented in Event-B. In [95, 96], M. Sarshogh and M. Butler introduce three Event-B trigger-response patterns, namely, deadlines, delays and expires, to encode discrete timing properties in Event-B. A “deadline” means that a set of events must respond to a particular event within a bounded time. For a “delay”, the set of response events must wait for a specified period after the triggering of an event. An “expiry” pattern prevents response events from triggering after the occurrence of an event. The authors translate timing properties as invariants, guards and Event-B actions. We are interested in investigating on how our code generation framework can be extended to support timing properties in Event-B, and in encoding this extension in EventB2Java once the Rodin platform fully supports the use of discrete timing events.

Chapter 9

Conclusion

This thesis investigated the answer to the question: is it possible to combine Refinement Calculus and Design-by-Contract together in the development of systems taking the best of both? For this purpose, this thesis presented the translation from Event-B models to JML-annotated Java code. The translation is defined by means of syntactic rules. These rules were implemented as the EventB2Java tool which is a Rodin plug-in (described in Chapter 5). EventB2Java bridges Refinement Calculus with Event-B to Design-by-Contract with JML and Java, answering the research question we proposed from the beginning: EventB2Java does allow users to combine Refinement Calculus and Design-by-Contract together in the development of systems enabling users to take advantage of the strengths and avoiding the weaknesses of each approach. EventB2Java generates Java executable code directly from abstract (or more refined) Event-B models, providing the option to verify the code against the generated JML specifications whenever users decide to customise the code. When modelling in Event-B, users need to prove the system to be consistent by discharging proof obligations. Rodin generates these proof obligations which some of them are automatically discharged by Rodin's provers and some others need the intervention of the user. The manually proving of proof obligations can be a difficult task, so we proposed a translation from Event-B proof obligations to the input language of Dafny, thus users can use Z3 (the automatic prover associated to Dafny) as a prover (described in Chapter 6). Our intention is to provide tools that help users in the process of proving an Event-B model sound.

The findings of this investigation could be of interest to both researchers working with Event-B and software developers working with JML and Java. Using Event-B in the early stages of the software development gives developers excellent support for modelling software systems in an abstract manner, and particularly for verifying safety, security and correctness properties of those models. Transitioning to JML at an appropriate point (as determined by the developers themselves, rather than being dictated by the tools being used) allows developers to take full advantage of data structures and APIs in the implementation language, and permits software developers with less mathematical expertise to contribute earlier in the development process. Furthermore, having the JML specifications embedded into the Java code also gives an insight of documentation of the code that can be read easily.

Researchers investigating about the translation from a language A to a language B might also find this thesis of interest. Researchers can see the development of this thesis as a guide for defining translation rules and its implementation from one language to another.

Code generation for Event-B is not a new concept. For instance, modelling in Event-B one can refine an Event-B model until accomplishing an actual implementation. However, this is difficult since users have to discharge proof obligations that become harder through refinements. This particular issue can make the use of formal methods less popular. Another way of code generation for Event-B is the use of existing tools. There are tools that allow users to translate Event-B models to different programming languages. Our tool outperforms these tools since: EventB2Java fully supports the Event-B syntax, except for non-deterministic assignments for which the tool generates JML specifications. Hence, EventB2Java does not impose restrictions on the syntax of the model to be translated, users can decide the level of abstraction in the model then transition to Java code. Current tools for generating Java code from Event-B models impose restrictions to users since the tools do not fully support the Event-B notation, so users need to evolve the Event-B model to use the syntax supported by those tools; on the other hand, as far as I can tell, our tool is the only one that generates JML specifications embedded in the Java code. This gives the user another layer of confidence when the user decides to customise the code since the generated code can be verified against the JML specifications.

In my experience on developing software without formal methods I have seen that one can end up with a final implementation of the software in a relatively short time. However, in many cases, the non use of formal methods makes the implementation misbehave so one needs to correct the implementation, making the maintenance of software a difficult task and a waste of time. For instance, since we are not using formal methods, reasoning about the model is not possible, so finding an error is difficult. One could use testing to uncover misbehaviours/erros, but testing can tell about problems among the scope of the testing process but nothing beyond. I have found very interesting the development of software using EventB2Java since the development starts in Event-B, where one can propose properties that the system needs to preserve and one must prove that the model indeed preserves those properties, for which EventB2Dafny can help in this process. Then, use the EventB2Java tool to generate Java code where the development can continue. The generated Java code is an actual implementation of the Event-B model and contains two main advantages: *i*) the code preserves the initial properties that the user defined, and *ii*) the code contains JML specifications so users can customise the code being able to check if the customised code meets the JML specifications. Users can argue that the use of Event-B can be difficult since one needs to discharge proof obligations delaying the development of software at early stages. However, my experience on using Event-B at early stages of development suggests that the final implementation of the development needs not to be maintained since the code does not contain errors. So, regarding times of development of software, the use of EventB2Java is better than using just Java or any other language.

I believe that having a tool that translates Refinement Calculus models into Design-by-Contract makes formal techniques and tools more usable. This is of paramount importance since the popularity of formal methods has not increased as much as researchers might want due to the level of expertise needed to work with these methods. Having tools that automise the process of working with these methods give the popularity formal method should have. I have seen the popularity of different things being increased by the development of tools. For instance, theorem proving was introduced by Begriffsschrift in 1884, but just until a couple of years ago theorem proving has become more popular thanks to the implementation of tools that assist users in the process. I see the use of Event-B and JML increasing thanks to EventB2Java since the tool enables people to automatise the process of combining Event-B and Java+JML in the development of software. EventB2Java still needs to be more developed, specially in proving the soundness of the entire translation rules, but I believe the investigation of this thesis is one step forward on making formal method more popular.

As a result of my study, further research might be to undertake a more complex software development using EventB2Java, involving different expertise researchers, experts in the notation underlying Event-B and Java-JML developers. An interesting case study is to implement the same software development using three approaches: using our tool, using just Java, and using just Event-B. So we could take metrics to compare the three developments to have an insight on the time to develop the software, on the effort put by modellers and implementers, and on the time used to maintain the software.

Another interest further research might be to use EventB2Java in Academia to help students to relate formal developments in Event-B with Java and JML. Nowadays, the use of formal methods is done by theoretic researchers, software developers are not so much familiarised with mathematics, logics, so they do not use formal methods. Hence, they are skeptical on their use. On the other hand, theoretic researchers do not use much programming languages. Both sides can argue about (dis-)advantages of each approach. It is quite difficult to convince any end to use another approach. I truly believe, the combination of methods like Event-B with Java and JML can be accomplished by teaching future software developers and theoretic researchers from their education. I see EventB2Java can fullfil this purpose since the tool enables students to relate formal methods and code in software development.

Bibliography

[1] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: A Java Constraint Kit. In M. Hanus, editor, *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] J.-R. Abrial. Sequential Program Development: Teaching Resources. `http://deploy-eprints.ecs.soton.ac.uk/122/1/sld.ch15%2Cseq.pdf`, 2009.

[4] J.-R. Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, New York, NY, USA, 2010.

[5] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 77(1,2):1–24, 2007.

[6] J.-R. Abrial, S. Schuman, and B. Meyer. Specification Language. In *On the Construction of Programs*, R. McKeag and A. Macnaghten, editors, pages 343–410. Cambridge University Press, 1980.

[7] B. Appleton. Patterns and software: essential concepts and terminology, 2000.

[8] Atelier B, the industrial tool to efficiently deploy the B Method, January 2008.

[9] R. Back and J. Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer, 1989.

[10] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[11] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*. Springer, 2006.

[12] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[13] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[14] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Verlag John Wiley & Sons, Inc, 1995.

[15] G. Bernot, M. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[16] Y. Bertot, P. Castran, G. Huet (informaticien), and C. Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004.

[17] R. Black. *Managing the Testing Process*. Wiley Publishing, Inc., 2009.

[18] J. Boulanger. ABTools: Another B Tool. In *Proceedings of Application of Concurrency to System Design*, 2003.

[19] J. Boulanger. B/HDL: design of safety circuit. In *Information and Communication Technologies*, volume 2, pages 2855 – 2860, 2006.

[20] J. Boulanger. The ABTools Suite. `http://sourceforge.net/projects/abtools/`, 2011.

[21] F. Bouquet, J. Couchot, F. Dadeau, and A. Giorgetti. Instantiation of Parameterized Data Structures for Model-Based Testing. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 96–110. Springer, January 2007.

[22] F. Bouquet, F. Dadeau, and J. Groslambert. Checking JML specifications with B machines. In *Proceedings of ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 435–454. Springer Verlag, April 2005.

[23] F. Bouquet, F. Dadeau, and J. Groslambert. JML2B: Checking JML Specifications with B Machines. In J. Julliand and O. Kouchnarenko, editors, *Proceedings of B: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 285–288. Springer Berlin / Heidelberg, 2006.

[24] J. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8:189–209, 1993.

[25] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005.

[26] L. Burdy and A. Requet. JACK: Java Applet Correctness Kit. In *In Proceedings, 4th Gemplus Developer Conference*, 2002.

[27] M. Butler. Decomposition Structures for Event-B. In *Proceedings of the 7th International Conference on Integrated Formal Methods*, IFM '09, pages 20–38, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors. *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, Lecture Notes in Computer Science. Springer, 2006.

[29] R. Butler. What is Formal Methods?, 2001. Available at `http://shemesh.larc.nasa.gov/fm/fm-what.html`.

[30] N. Cataño. The EventB2Dafny Tool, 2012. Available at `http://poporo.uma.pt/eventb2dafny/index.html`.

[31] N. Cataño and M. Huisman. Chase: A Static Checker for JML's Assignable Clause. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proceedings of Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40, New York, NY, USA, January 9-11 2003. Springer-Verlag.

[32] N. Cataño, J. Pestana, and R. Rodrigues. JFly: A JML-Based Strategy for Incorporating Formal Specifications into the Software Development Process. In Luis Barbosa and Miguel P. Correia, editors, *Portuguese forum of Informatics*, Braga, Portugal, September 2010.

[33] N. Cataño and V. Rivera. The B2Jml Tool, 2012. Available at `http://poporo.uma.pt/favas/b2jml.html`.

[34] N. Cataño and V. Rivera. The EventB2Jml Tool, 2012. Available at `http://poporo.uma.pt/favas/eventb2jml.html`.

[35] N. Cataño, V. Rivera, and R. Leino. The EventB2Dafny Rodin Plug-In. In *2nd Workshop on Developing Tools as Plug-ins (TOPI)*, pages 49–54, Zurich, Switzerland, June 3 2012. IEEE Xplore.

[36] N. Cataño, V. Rivera, C. Rueda, and T. Wahls. *Formal Methods Applied to Industrial Complex Systems: Implementation of the B Method*, chapter Translating B and Event-B Machines to Java and JML, pages 211–252. Computer Engineering Series. Wiley, July 2014.

[37] N. Cataño and C. Rueda. Teaching Formal Methods for the Unconquered Territory. In *2nd International Formal Methods Europe Conference on Teaching Formal Methods*, Lecture Notes in Computer Science, The Netherlands, 2009. Springer-Verlag.

[38] N. Cataño and C. Rueda. Matelas: A Predicate Calculus Common Formal Definition for Social Networking. In M. Frappier, editor, *Proceedings of ABZ 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 259–272, Québec, Canada, 2010. Springer Berlin Heidelberg.

[39] N. Cataño, C. Rueda, and T. Wahls. A Machine-Checked Proof for a Translation of Event-B Machines to JML. *CoRR*, abs/1309.2339, 2013.

[40] N. Cataño and T. Wahls. Executing JML Specifications of Java Card Applications: A Case Study. In *24th ACM SAC, Software Engineering Track*, Waikiki Beach, Honolulu, Hawaii, 2009.

[41] N. Cataño, T. Wahls, C. Rueda, V. Rivera, and Danni Yu. Translating B Machines to JML Specifications. In *27th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*, Trento, Italy, March 26-30 2012.

[42] A. Cavalcanti, F. Zeyda, A. Wellings, J. Woodcock, and K. Wei. Safety-critical Java programs from Circus models. *Real-Time Systems*, 49(5):614–667, 2013.

[43] Y Cheon and G. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the international conference on Software Engineering Research and Practice , Las Vegas*, pages 322–328. CSREA Press, 2002.

[44] Choco Team. Choco: an open source Java constraint programming library. Research report 10-02-INFO, École des Mines de Nantes, 2010.

[45] D. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods Symposium*, pages 472–479, 2011.

[46] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML - Progress and issues in building and using ESC/Java2. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop*. SpringerVerlag, 2004.

[47] Ada Core. Tokeneer. `http://www.adacore.com/sparkpro/tokeneer/download`, 2013.

[48] K. Damchoom. *An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B*. PhD thesis, University of Southampton, 2010.

[49] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[50] D. Dean. eLoc Engine. Accessed September 2013. `http://sourceforge.net/projects/elocengine/`, (2013).

[51] D. Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 2011. Available online.

[52] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, August 1975.

[53] P. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[54] A. Edmunds and M. Butler. Tool Support for Event-B Code Generation. In *Workshop on Tool Building in Formal Methods*, Québec, Canada, 2010. J. Wiley and Sons.

[55] A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, 2011.

[56] A. Edmunds and A. Rezazedah. Development of a Heating Controller System, 2011. Available at `http://wiki.eventb.org/index.php/Development_of_a_Heating_Controller_System`.

[57] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[58] J. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, 2003.

[59] J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermann, editors, *Conference on Computer-Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, 2007.

[60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. A. Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[61] Google Inc. The Android Platform. `http://developer.android.com/design/index.html`, 2012.

[62] D. Jin and Z. Yang. Strategies of Modeling from VDM-SL to JML. In *International Conference on Advanced Language Processing and Web Information Technology*, pages 320–323, 2008.

[63] C. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, second edition, 1990.

[64] The KeY Project, Integrated Deductive Software Design. `http://www.key-project.org/`.

[65] J. Koenig and K. Leino. Getting Started with Dafny: A Guide. In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series – D: Information and Communication Security*, pages 152–181. IOS Press, May 2012.

[66] B. Krause and T. Wahls. jmle: A Tool for Executing JML Specifications via Constraint Programming. In *Proceedings of FMICS*, volume 4346 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[67] B. Krause and T. Wahls. jmle: A Tool for Executing JML Specifications via Constraint Programming. In *Proceedings of Formal Methods for Industrial Critical Systems*, volume 4346 of *Lecture Notes in Computer Science*, Bonn, Germany, 2006. Springer-Verlag.

[68] K.Traichaiyaporn. Modeling Correct Safety Requirements Using KAOS and Event-B. Master's thesis, Japan Advanced Institute of Science and Technology, 2013.

[69] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.

[70] G. Leavens, A. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM Special Interest Group Software Engineering*, 2006.

[71] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. Zimmerman, and W. Dietl. JML Reference Manual. `http://www.eecs.ucf.edu/~leavens/JML/OldReleases/jmlrefman.pdf`, May 2013.

[72] M. Leuchel and M. Butler. ProB: A Model Checker for B. In *Formal Methods Europe*, Lectures Notes in Computer Science 2805. Springer-Verlag, 2003.

[73] J. Link. *Unit Testing in Java*. M. Kaufmann, 2003.

[74] D. Locke, B. Andersen, B. Brosgoal, M. Fulton, T. Henties, J. Hunt, J. Nielsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Weillings. Safety Critical Java Specification, version 0.78. Technical report, The Open Group, `http://jcp.org/aboutJava/communityprocess/edr/jsr302/index.html`, 2010.

[75] C. Marche, C. Paulin-mohring, and X. Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML, 2003.

[76] D. Mentré, C. Marché, J. Filliâtre, and M. Asuka. Discharging Proof Obligations from Atelier B using Multiple Automated Provers. In *ABZ Conference*, Pisa, Italy, June 2012.

[77] D. Méry and N. Singh. Automatic code generation from Event-B models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT. ACM, 2011.

[78] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[79] C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[80] J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.

[81] L. De Moura and N. Bjorner. Z3: An Efficient Solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`, 2010.

[82] J. Nielsen. *Usability Engineering*. AP Professional, San Diego, CA, USA, 1993.

[83] S. Ostroumov and L. Tsiopoulos. VHDL Code Generation from Formal Event-B Models. In *Proceedings of the 14th Euromicro Conference on Digital System Design*, Euromicro Conference on Digital System Design, pages 127–134, Washington, DC, USA, 2011. IEEE Computer Society.

[84] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[85] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI, 2006.

[86] S. Padidar. A Study In The Use Of Event-B For System Development From A Software Engineering Viewpoint. Master's thesis, University of Edinburgh, 2010.

[87] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[88] L. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[89] D. Peled, D. Gries, and F. Schneider, editors. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[90] V. Rivera and N. Cataño. The Social-Event Planner, 2012. Available at `http://poporo.uma.pt/favas/Social-Event_Planner.html`.

[91] V. Rivera and N. Cataño. Translating Event-B to JML-Specified Java programs. In *29th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*, Gyeongju, Korea, March 24-28 2014.

[92] V. Rivera, N. Cataño, T. Wahls, and C. Rueda. Code Generation for Event-B. Submitted to Software Tools for Technology Transfer, 2014.

[93] Rigorous Development of Complex Fault-Tolerant Systems. Accessed February 2012. `http://sourceforge.net/projects/rodin-b-sharp/`, (2011).

[94] Rodin Developer Support. Accessed February 2012. `http://wiki.event-b.org/index.php/Rodin_Developer_Support`, (2011).

[95] M. Sarshogh and M. Butler. Specification and refinement of discrete timing properties in Event-B. *Electronic Communications of the European Association of Software Science and Technology*, 46, 2011.

[96] M. Sarshogh and M. Butler. Extending Event-B with Discrete Timing Properties. `http://deploy-eprints.ecs.soton.ac.uk/401/1/Journal.pdf`, 2012.

[97] State-Machines and Code Generation. Accessed August 2013., 2012. Available at `http://wiki.event-b.org/index.php/State-Machines_and_Code_Generation`.

[98] A. Toom, T. Naks, M. Pantel, M. Gandriau, and Indrawati. Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos. In Akadeemia IB Krates OÜ, University of Toulouse IRIT-ENSEEIHT, F. Alyotech CRIL Technologies, and Tallinn University of Technology, editors, *Embedded Real Time Software*, 2008.

[99] J. van den Berg and B. Jacobs. LOOP compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312. Springer, 2001.

[100] L. Voisin. Description of the Rodin Prototype: RODIN Deliverable D15. `http://rodin.cs.ncl.ac.uk/deliverables/rodinD15.pdf`, 2006.

[101] A. Wellings. *Concurrent and Real-Time Programming in Java.* J. Wiley & Sons, 2004.

[102] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* International Series in Computer Science. Prentice-Hall, Inc., 1996.

[103] S. Wright. Automatic Generation of C from Event-B. In *Workshop on Integration of Model-based Formal Methods and Tools.* Springer-Verlag, 2009.

[104] F. Yang, J. Jacquot, and J. Souquières. The Case for Using Simulation to Validate Event-B Specifications. In *APSEC*, pages 85–90, 2012.

[105] D. Zimmerman and R. Nagmoti. JMLUnit: The Next Generation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, FoVeOOS'10, pages 183–197, Berlin, Heidelberg, 2011. Springer-Verlag.

Appendix A

Event-B syntax supported by EventB2Java

This appendix shows the full syntax of Event-B and the translation to Java and JML. The Event-B modelling language is composed of five mathematical languages (see Chapter 9 of [4]), namely, *a*) a Propositional Language, *b*) a Predicate Language, *c*) an Equality Language, *d*) a Set-Theoretic Language, and *e*) Boolean and Arithmetic Languages. The following shows the translation of each construct in the Event-B languages to Java and JML.

P and Q are predicates, E and F are expressions, x is a variable, S and T are sets, f and g are relations, Pr is a Pair, and a and b are Integers. The construct `Type(tt)` translates the type of the Event-B variable tt to the corresponding Java type.

A.1 The Propositional Language

Event-B Op.	JML	Java
$\neg P$	`!P`	same as JML
$P \wedge Q$	`P && Q`	same as JML
$P \vee Q$	`P || Q`	same as JML
$P \Rightarrow Q$	`BOOL.implication(P,Q)`	same as JML
$P \Leftrightarrow Q$	`BOOL.bi_implication(P,Q)`	same as JML

A.2 The Predicate Language

Event-B Op.	JML	Java
$\forall x \cdot P$	`(\forall Type(x) x ; P)`	not supported yet!
$\exists x \cdot P$	`(\exists Type(x) x ; P)`	not supported yet!
$E \mapsto F$	`new Pair(E,F)`	same as JML
$E = F$	`E.equals(F)`	same as JML

A.3 The Set-Theoretic Language

Event-B Op.	JML	Java
$E \in F$	`F.has(E)`	same as JML

A.3.1 Axioms of set theory

Event-B Op.	JML	Java
$E \times F$	`BRelation.cross(E,F)`	same as JML
$\mathbb{P}(E)$	`E.pow()`	same as JML
$\{x \cdot P \mid F\}$	`new BSet<Type(x)>(new JMLObjectSet Type(e) e | (\exists Type(x) x; P; e.equals(E)))`	not supported yet!
$E = F$	`E.equals(F)`	same as JML

A.3.2 Elementary set operators

Event-B Op.	JML	Java
$S \subseteq T$	`S.isSubset(T)`	same as JML
$S \subset T$	`S.isProperSubset(T)`	same as JML
$S \cup T$	`S.union(T)`	same as JML
$S \cap T$	`S.intersection(T)`	same as JML
$E \setminus T$	`S.difference(T)`	same as JML
$\varnothing$	`BSet.EMPTY`	same as JML

A.3.3 Generalisation of elementary set operators

Event-B Op.	JML	Java
$\mathit{union}(S)$	`BSet.union(S)`	same as JML
$\bigcup x \cdot P \mid E$	`BSet.union(new BSet<Type(x)>(new JMLObjectSet Type(e) e | (\exists Type(x) x; P; e.equals(F))))`	not supported yet!
$\mathit{inter}(S)$	`BSet.intersection(S)`	same as JML
$\bigcap x \cdot P \mid E$	`BSet.intersection(new BSet<Type(x)>(new JMLObjectSet Type(e) e | (\exists Type(x) x; P; e.equals(F))))`	not supported yet!

A.3.4 Binary relation operators

Event-B Op.	Java	JML
$S \leftrightarrow T$	`BRelation<Type(S),Type(T)>`	*
$S \trel T$	`BRelation<Type(S),Type(T)>`	*
$S \srel T$	`BRelation<Type(S),Type(T)>`	*
$S \strel T$	`BRelation<Type(S),Type(T)>`	*
$dom(f)$	`f.domain()`	same as Java
$range(f)$	`f.range()`	same as Java
f^{-1}	`f.inverse()`	same as Java
$S \lhd f$	`f.restrictDomainTo(S)`	same as Java
$S \rhd f$	`f.restrictRangeTo(S)`	same as Java
$S \domsub f$	`f.domainSubtraction(S)`	same as Java
$S \ransub f$	`f.rangeSubtraction(S)`	same as Java
$f[x]$	`f.image(x)`	same as Java
$f \ ; \ g$	`f.compose(g)`	same as Java
$f \circ g$	`f.backwardCompose(g)`	same as Java
$f \ovl g$	`f.override(g)`	same as Java
$f \otimes g$	`f.directProd(g)`	same as Java
$f \parallel g$	`f.parallel(g)`	same as Java

$*$ JML specifications associate to this operator is explained in A.5

A.3.5 Functions operators

Event-B Op.	Java	JML
id	`new ID()`	same as JML
$S \pfun T$	`BRelation<Type(S),Type(T)>`	*
$S \rightarrow T$	`BRelation<Type(S),Type(T)>`	*
$S \pinj T$	`BRelation<Type(S),Type(T)>`	*
$S \rightarrowtail T$	`BRelation<Type(S),Type(T)>`	*
$S \psur T$	`BRelation<Type(S),Type(T)>`	*
$S \twoheadrightarrow T$	`BRelation<Type(S),Type(T)>`	*
$S \tbij T$	`BRelation<Type(S),Type(T)>`	*
$prj_1(Pr)$	`Pr.fst()`	same as Java
$prj_2(Pr)$	`Pr.snd()`	same as Java

$*$ JML specifications associate to this operator is explained in A.5

A.4 Boolean and Arithmetic Language

Event-B Op.	JML	Java
BOOL	`BOOL`	same as JML
TRUE	`true`	same as JML
FALSE	`false`	same as JML
$\mathbb{Z}$	`INT`	same as JML
$\mathbb{N}$	`NAT`	same as JML
$\mathbb{N}1$	`NAT1`	same as JML
$succ(a)$	`a+1`	same as JML
$pred(a)$	`a-1`	same as JML
0	`0`	same as JML
1	`1`	same as JML
...		
$a + b$	`a + b`	same as JML
$a * b$	`a + b`	same as JML
$a \,\hat{}\, b$	`Math.pow(a,b)`	same as JML

A.4.1 Extension of the arithmetic language

Event-B Op.	JML	Java
$a \leq b$	`a.compareTo(b) <= 0`	same as JML
$a < b$	`a.compareTo(b) < 0`	same as JML
$a \geq b$	`a.compareTo(b) >= 0`	same as JML
$a > b$	`a.compareTo(b) > 0`	same as JML
$finite(S)$	`S.finite()`	same as JML
$a \mathinner{..} b$	`new Enumerated(a,b)`	same as JML
$a - b$	`a - b`	same as JML
a/b	`a / b`	same as JML
$a \ mod \ b$	`a % b`	same as JML
$card(S)$	`S.size()`	same as JML
$max(S)$	`S.max()`	same as JML
$min(S)$	`S.min()`	same as JML

A.5 Some other JML specs

All Event-B functions and relations are translated as instances of `BRelation`. Restrictions made by each kind of function/relation are translated to the JML `invariant`:

Event-B Op.	**JML spec**
$f \in S \leftrightarrow T$	`f.domain().isSubset(S) &&` `f.range().isSubset(T)`
$f \in S \trel T$	`f.domain().equals(S) &&` `f.range().isSubset(T)`
$f \in S \srel T$	`f.domain().isSubset(S) &&` `f.range().equals(T)`
$f \in S \strel T$	`f.domain().equals(S) &&` `f.range().equals(T)`

$f \in S \pfun T$	`f.isaFunction() &&`
	`f.domain().isSubset(S) &&`
	`f.range().isSubset(T)`

$f \in S \to T$	`f.isaFunction() &&`
	`f.domain().equals(S) &&`
	`f.range().isSubset(T)`

$f \in S \pinj T$	`f.isaFunction() &&`
	`f.inverse().isaFunction() &&`
	`f.domain().isSubset(S) &&`
	`f.range().isSubset(T)`

$f \in S \rightarrowtail T$	`f.isaFunction() &&`
	`f.inverse().isaFunction()`
	`&& f.domain().equals(S) &&`
	`f.range().isSubset(T)`

$f \in S \psur T$	`f.isaFunction() &&`
	`f.domain().isSubset(S) &&`
	`f.range().equals(T)`

$f \in S \twoheadrightarrow T$	`f.isaFunction() &&`
	`f.domain().equals(S) &&`
	`f.range().equals(T)`

$f \in S \tbij T$	`f.isaFunction() &&`
	`f.inverse().isaFunction()`
	`&& f.domain().equals(S) &&`
	`f.range().equals(T)`