A Study In The Use Of Event-B For System Development From A Software Engineering Viewpoint

MSc Dissertation

Author: Sasan Padidar

Supervisors: Dr. Gudmund Grov, Prof. Alan Bundy
## Contents

1 Introduction ................................................................. 1
   1.1 Motivation .............................................................. 1
   1.2 Project Plan ............................................................ 3
      1.2.1 Learning Event-B & Rodin ..................................... 3
      1.2.2 The Real World Case Study .................................. 4
      1.2.3 Initial Work-plan ............................................... 4
      1.2.4 Changes ........................................................... 5
   1.3 Outline of Thesis ...................................................... 6
      1.3.1 Contributions .................................................... 6

2 Background ................................................................. 8
   2.1 Approaches To Verified Software .................................... 8
      2.1.1 Correctness By Construction .................................. 8
      2.1.2 Source-Code Verification ...................................... 9
   2.2 Z Notation ............................................................... 9
      2.2.1 Z Basics .......................................................... 9
   2.3 Tokeneer ................................................................. 10
      2.3.1 Tokeneer In Z .................................................... 12
      2.3.2 Specification Document ........................................ 13
   2.4 Reasoned Modelling Patterns ....................................... 15
   2.5 Absolute Block Signalling .......................................... 15
   2.6 Summary ................................................................. 16

3 Event-B & Rodin ........................................................... 17
   3.1 Event-B ................................................................. 17
      3.1.1 Refinement ....................................................... 18
      3.1.2 Decomposition ................................................... 18
      3.1.3 Generic Development .......................................... 18
      3.1.4 Modelling in Event-B .......................................... 18
      3.1.5 Event-B Proof Obligations .................................... 19
   3.2 ABS Model ............................................................... 20
      3.2.1 Definitions ...................................................... 20
      3.2.2 The Protocol ..................................................... 21
      3.2.3 ABS Event-B Model ............................................ 22
      3.2.4 ABS Proof Obligations ........................................ 26
   3.3 Rodin .................................................................... 27
      3.3.1 Basics ............................................................ 27
      3.3.2 Modelling Perspective ......................................... 27
      3.3.3 Proof Perspective ............................................... 27
   3.4 Summary ............................................................... 29

4 Modelling Tokeneer ..................................................... 31
   4.1 Introduction ............................................................ 31
      4.1.1 Modelling Strategy .............................................. 31
      4.1.2 Strategy Modifications ....................................... 32
      4.1.3 Difficulties In Modelling ..................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.4</td>
<td>Using Refinement</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Tokeneer Model</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Conversion of Z to Event-B</td>
<td>34</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Carrier-sets &amp; Schemas</td>
<td>34</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Conversion Decisions</td>
<td>35</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Optional Keyword</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>The Model</td>
<td>35</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Time</td>
<td>36</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Tokens</td>
<td>36</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Certificates</td>
<td>37</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Current Token</td>
<td>37</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Roles</td>
<td>38</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Operations</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>Patterns &amp; Guidance</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Initialisation</td>
<td>41</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Examples</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Numbers</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Evaluation</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Analysis of Formal Methods</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Difficulties In Modelling</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>Modelling and Implementation</td>
<td>47</td>
</tr>
<tr>
<td>6.4</td>
<td>Mapping Specification to Implementation</td>
<td>48</td>
</tr>
<tr>
<td>6.5</td>
<td>The Idea of Refinement</td>
<td>48</td>
</tr>
<tr>
<td>6.6</td>
<td>Measuring Quality</td>
<td>49</td>
</tr>
<tr>
<td>6.7</td>
<td>Role of Experience</td>
<td>50</td>
</tr>
<tr>
<td>6.8</td>
<td>Undecidable Logic</td>
<td>50</td>
</tr>
<tr>
<td>6.9</td>
<td>Rodin Experience</td>
<td>51</td>
</tr>
<tr>
<td>6.9.1</td>
<td>Advantages</td>
<td>51</td>
</tr>
<tr>
<td>6.9.2</td>
<td>Disadvantages</td>
<td>52</td>
</tr>
<tr>
<td>6.10</td>
<td>Summary</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>55</td>
</tr>
<tr>
<td>7.1</td>
<td>Future Work</td>
<td>56</td>
</tr>
<tr>
<td>A</td>
<td>Tokeneer Context</td>
<td>59</td>
</tr>
<tr>
<td>B</td>
<td>Tokeneer Machine</td>
<td>63</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to express my sincerest gratitude to my supervisors, Dr Gudmund Grov and Professor Alan Bundy, without whom this project would not have been completed. Their guidance and efforts during the course of this project have been invaluable. Especially, I would like to thank them for giving me the opportunity to be a participant at the Verified Software: Theories, Tools and Experiments (VSTTE) 2010 conference. The understanding that I have gained in my endeavours in learning to create reliable software, is indebted to them.

A special thanks goes to NSF for sponsoring my registration fees for the VSTTE conference.
Abstract

Inexperienced users, when approaching formal methods, encounter a number of challenges and ambiguities that could discourage them from using formal methods. In this study we try to demonstrate the issues that surround inexperienced users when dealing with formal methods and for some of these issues, suggestions are made on how to overcome them. We model a real-world case study in Event-B which is a formal method used to model and reason about discrete systems. Event-B follows the "posit and prove" approach, where a user posits a specification and justifies the system by proving the proof obligations. In some cases Event-B cannot prove some proof obligations automatically and it requires expert human interaction to complete the proof. For some of these cases we also suggest some solutions which could guide novice users to discharge the proof obligations.
Chapter 1

Introduction

In this chapter we state the motivation for using formal methods and we state our reasons about why formal methods should become more widely accepted. Our aim is to show the importance of formal methods and the difference that they can make to improve software development. Hence justify our purpose for finding patterns.

1.1 Motivation

Making mistakes in programs is inevitable and therefore it is absolutely necessary to have some tools and techniques to identify the mistakes and deal with them in advance [18]. This is not purely because of the level of competency of programmers, as the experienced programmer makes as many mistakes as the novice programmers (although the context of mistakes might differ) [18]. Production of reliable software has been an expensive and also a challenging problem in software production. This difficulty has driven scientists to develop different methods to make this task easier and more effective [20]. Development of software by the traditional methods of quality assurance and testing is both unreliable and error prone as they cannot check all possible cases to ensure the correctness of software. Also, tests are written by humans and the possibility of testers following the same line of reasoning as the programmers is quite likely [18]. Thus testing programs can only suggest that the tests produce the desired result and this does not prove any property of the actual system.

Regan et al. [18] explain that the programs that are produced in industry are far more complex that can be grasped easily. Thus it is also extremely difficult to check them as you need to understand the systems in order to be able to test them. Therefore it is necessary to have a logical and mathematically accepted method to allow us to specify and model software systems in a mathematically sound way [2]. However, there is overwhelming evidence of the need for a mature method of tackling software problems. A strategy that can provide some kind of guarantee that a software product can be trusted to be functional, according to its requirements. Note that we are not suggesting that these methods are currently in a state that can be used easily by an inexperienced user in the field of formal methods.

In order to develop software engineering further and transform it from its current state, we need to incorporate engineering disciplines, the kind of disciplines that civil engineers have in their field. What this means is that we need to have widely accepted methods for creating reliable software. However, such methods do not exist to a great extent. There are "rules-of-thumb" that software engineers use based on their level of experience, although their usefulness can be questioned.
We understand that all programs do not require the same level of integrity and safety, but that does not mean the idea and the strategy that formal methods offer should be ignored for those programs. For instance, as the need for concurrent programs increases, the more inadequacies of traditional software testing methods become apparent because we cannot test a concurrent system as each run is very likely to be different than the previous. The only method that is used to test such systems is by running the program a number of times and check the flow of the program sequentially. Such difficulties are known to be extremely challenging to deal with.

It might be discouraging to use formal methods because of their sheer difficulty for users with less mathematical background but the truth about the immaturity of software development processes in design and correctness need to be addressed in order to mature software development.

A promising approach to tackle software production is formal methods. Formal methods are mathematical techniques that are supported by different tools to help software or hardware production [20] in modelling and specifying their properties in a mathematical way which can be proved to verify the correctness of the software or hardware. One major example of using formal methods for software verification and quality assurance is NASA. The investment in research and development of tools that NASA has done to produce reliable software for its missions have shown that using formal methods for software verification is both practical and significantly reliable [18]. This result suggests that using formal methods for producing reliable software can provide some certainty in the proper functionality of the final program.

One success story of formal methods, namely classical B [1] method was used for the Paris Metro Line 14. The size of this project was quite large. The number of lines of B was 115,000 lines and the number of proofs was 27,800. For this project, no unit tests were ever written which means the verification process was justified by proving the correctness of the system that was designed [20]. Only a number of global system tests were performed to demonstrate the functionality of the complete system.

The industrial establishments that currently use formal methods are, mainly, using them in the specification and design phase of software production [20]. At the implementation level, formal methods are used for code verification. Every program specification asserts a correctness theorem that, if some conditions hold, the program will achieve the described result that the documentation specifies [20]. Nearly 60% of projects in industry that have used mathematical techniques for specifying software, have used specification and modelling as the primary method of producing reliable software [20].

The use of formal methods in creating reliable software is now a necessity for safety critical systems [18]. It is also essential to have tools that automate the process as much as possible to reduce the complexities and also increase the speed of software production and reduce costs [2]. This means that having less human interaction and more automation in proving the correctness of software products is highly desirable.

Another major example of use of formal methods is the Tokeneer system that was developed on an industrial level and demonstrated that it is possible to use formal methods to ensure the correctness of the system being modelled. One important point about this project was that it showed the final system required significantly less effort to maintain compared to other systems that do not use such methods [6]. In this project since its delivery in 2003, only two software defects were found which is a remarkable success. This means, formal methods can be an appropriate choice for specifying software that requires high quality standards.
It is worth noting that the assumption that creation of reliable software costs more and requires more effort is not necessarily true. Because, according to [20], the cost of these projects might be higher at the start of the project but in the long term it will not cost as much as the maintenance of the traditional software methodologies. There might be another question which is "is formal methods the right approach to use for software production?". The answer to this question is "yes" as those industrial software productions which have used mathematical techniques for specifying (modelling) their software, have used formal methods and have achieved remarkable results as shown in [20].

Therefore based on the reasons provided above, formal methods can be a reasonable approach to tackle the challenges of software production. So automating more and simplifying formal methods can be a significant step towards attracting more users to this field. One approach to automate more and simplify this process is introducing patterns and guidelines in order to reduce human interaction with tools i.e. provers to expedite the process of software verification. The AI4FM project \footnote{www.ai4fm.org} claims that it might be able to automate the proving process by learning from one manual proof to automate the rest of similar proofs that exist in the model that is being specified. This project is an example of efforts that are being undertaken to simplify the use of formal methods for users with less mathematical background.

Our aim in this project is to demonstrate the challenges and difficulties which could make formal methods discouraging for a novice user. Also we aim to provide suggestions on overcoming these challenges by providing guidelines on methods that could be used to discharge proof obligations either by interactive theorem proving or modifying the model to simplify proof obligations. We know that experts in the field know the strategies for writing correct software but, the practicality of such strategies for a novice user is questionable. Thus by identifying these challenges we could help novice users to make an informed decision on whether to use formal methods. Our strategy to tackle this project is by modelling a large case study in Event-B and record every action taken to complete the project.

### 1.2 Project Plan

In this section we describe the entire plan that was set initially for approaching this project. Also, we describe the reasons for the instances where the plan had to be changed as a result of some setbacks which were mainly natural changes due to the nature of the project (i.e. we found reasoned modelling patterns instead of proof patterns). Moreover, the chosen strategy to overcome these challenges is described in detail.

#### 1.2.1 Learning Event-B & Rodin

To be able to complete this project, the main skill that is required is understanding Event-B and learning how to use Rodin. There are a number of tutorials and documents about how to use Event-B and Rodin, but we decided to learn them by modelling a very small case study which could be used as a tutorial of Event-B in the thesis. The chosen case study for this purpose is the "Absolute Block Signalling (ABS)" system \footnote{12}. This system is explained in more detail in section 2.5.
1.2.2 The Real World Case Study

In order to be able to find patterns and provide evidence of such patterns it is essential:

- to develop a model that, at least, has some connections to the real-world,
- and is complex enough that can increase the chance of generating interesting and relevant proof obligations (explained in more detail in chapter 3).

The Tokeneer system, as stated before, is a successful industrial project that used formal methods extensively. This project has a number of advantages to be used in this project:

- It is a real-world project that was specified and implemented by some of the major formal methods experts [6].
- Its specification uses the Z notation which is clear and understandable.
- All the documents and source code of Tokeneer is available online for the public.
- As the system is already built and modelled, the focus of the project can be directed to finding patterns rather than starting the modelling and requirements analysis from scratch.

Because of these reasons, the decision was made to use the Tokeneer system as the main case study of this project. This system is described in section 2.3 in more detail.

1.2.3 Initial Work-plan

The original plan of this project is provided below. The plan shows the initial work-packages that were set to be completed in a period of, approximately, three months. Overall, the project did follow the plan as was initially intended, however, changes were made to the project and its focus slightly, during the course of the project.

- **WP1**: Develop an understanding of Event-B and Rodin. Learning Event-B by modelling the "absolute block signalling" model. The ABS system is a model developed for ensuring trains will not collide. This model is rather simple and useful to be modelled to show the processes that are involved in building an Event-B model.

  **Deliverables:**
  - D1: A small case study done in Event-B using Rodin.
  - D2: Description of Event-B and Rodin.

- **WP2**: Understanding Tokeneer and identifying subsystem(s) of it. By learning more about Tokeneer and understanding the detailed operations involved, the decision about which segments to implement was made and the project was based on that decision.

  **Deliverables:**
  - D3: A description of Tokeneer.
  - D4: A document describing the parts that are modelled.
• **WP3:** Develop Tokeneer’s chosen segments.
  **Deliverables:**
  - D5: A formally verified Event-B model of the subsystem.
  - D6: A description of the subsystem and the modelling of it (chapter of the thesis)

• **WP4:** Extract patterns. Investigating similar patterns that might occur in the POs.
  **Deliverables:**
  - D7: A chapter on patterns found.
  - D8: An analysis describing where pattern (i.e. families of POs) typically occur. This analysis is done by categorising proofs and the found patterns. This categorisation can show the possibility of having the found pattern for a proof in the category can be applied to the other proofs in the same category.

• **WP5:** Extending the model if necessary. If there is extra time, modelling an optional segment of Tokeneer. This workpackage is added as a contingency plan as there might be problems during the course of the project i.e. lack of data. By having this workpackage we can have some certainty that these problems can be solved, in case they occur.
  **Deliverables:**
  - D9: A formally verified Event-B model of the subsystem.
  - D10: A description of the subsystems and the modelling of them.

• **WP6:** Thesis. Completion of the thesis by combining the background materials and all the gathered results.
  **Deliverables:**
  - D11: The thesis.

Note that workpackage 1 and 2 run in parallel, the Gantt chart shown in figure 1.2 (the durations are shown in the table in figure 1.1). Workpackage 5 is depended on workpackage 3 and workpackage 4 runs in parallel with both of them. Workpackage 6 is done after the main workpackages are completed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Start</th>
<th>Finish</th>
<th>Work</th>
<th>Duration</th>
<th>Slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>WP1</td>
<td>May 21</td>
<td>Jun 7</td>
<td>11d</td>
<td>12d</td>
<td>50d</td>
</tr>
<tr>
<td>WP2</td>
<td>May 21</td>
<td>Jun 7</td>
<td>12d</td>
<td>12d</td>
<td>50d</td>
</tr>
<tr>
<td>WP3</td>
<td>Jun 8</td>
<td>Jul 8</td>
<td>2/2d 5h</td>
<td>2/2d 5h</td>
<td>13d 2h</td>
</tr>
<tr>
<td>WP5</td>
<td>Jul 9</td>
<td>Jul 28</td>
<td>13d 6h</td>
<td>13d 6h</td>
<td>13d 1h</td>
</tr>
<tr>
<td>WP4</td>
<td>Jun 15</td>
<td>Jul 28</td>
<td>31d 5h</td>
<td>31d 5h</td>
<td>13d 2h</td>
</tr>
<tr>
<td>WP6</td>
<td>Jul 20</td>
<td>Aug 16</td>
<td>20d</td>
<td>20d</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: Durations

1.2.4 Changes

There are a number of changes that were made to this project during its course.
• Our initial approach of modelling was changed and it is explained in detail in §4.
• The focus from finding proof patterns was shifted to reasoned modelling patterns (combination of patterns from modelling and reasoning) [13], because of the nature of the project and the approach that was chosen. It seemed more relevant and useful for novice users to provide them some reasoned modelling patterns rather than proof patterns.
• The segments ofTokeneer that were modelled were not all formally verified. Because of lack of time, not all undischarged proof obligations were interactively proved.
• We also found it necessary to give an evaluation of the use of Event-B and Rodin. In particular, since the background of the author is in software engineering and convincing software engineers to use formal methods is a key challenge for wide acceptance of formal methods.

### 1.3 Outline of Thesis

- **Chapter 1** describes the problem and describes the use of formal methods in the industry.
- Chapter 2 provides the background knowledge required to understand the work that is done.
- Chapter 3 describes Event-B and Rodin to a level necessary to understand the project.
- Chapter 4 discusses the modelling process and the developed model.
- Chapter 5 is the description of the found patterns.
- Chapter 6 is an evaluation of the practicality of formal methods for a novice user and analysis of the use of Rodin.
- Chapter 7 is the overall conclusion of the project.

#### 1.3.1 Contributions

The contributions that this project has made are:
• Providing guidance for new users on the important factors that need to be considered and thought about carefully before choosing formal methods for their software production.

• Analysis of the practicality of Rodin and formal methods for new users and explaining the factors that could, potentially, be the reasons that are stopping formal methods to be used for general software development.

• Formulation and verification of segments of Tokeneer to Event-B.

• Identification of some patterns (more reasoned modelling patterns (described in section 2.4)).

• Develop a small case study to illustrate the use of Event-B and Rodin.

Note that this result was presented at poster session of VSTTE conference 16-19 August 2010 [17].
Chapter 2

Background

In this section the Z notation and the Tokeneer system are explained to the level necessary to understand the context of the project. Also reasoned modelling patterns are described to familiarise the reader with this kind of pattern. An overall introduction to more common practices of software verification is also provided.

2.1 Approaches To Verified Software

Verifying software contrary to the common belief, is not just summarised in theorem proving approaches. There are other methods such as model checking, correctness by construction approaches and static checkings that are categorised as formal methods. In this section we describe some of such methods.

2.1.1 Correctness By Construction

Programming languages have improved in three main categories, abstraction, strong type checking and structure. However, they have not evolved in their accuracy of expression [7]. As a result of this, the predictability of the compiled code has decreased. For instance, evolution of C to C++, undoubtedly, provided the ability to express design abstractions but, the predictability of the compiled code decreased.

This lack of precision in programming languages makes it difficult to reason about programs at an early stage [4]. What correctness by construction suggests is that, by having a programming language that has precise meaning, we can stop introducing errors as early as possible. This means building a language whose source code defines the behaviour of the compiled code i.e. no ambiguities exist in the source code. For example the Spark language \(^1\) is a programming language with a development platform which is an extension of the Ada programming language. The Spark language is precise and it requires programmers to think carefully and express themselves accurately. If there exist an error or some lack of precision in the source code, the Spark tool notifies the programmer to change the code accordingly. Compared to other programming languages, it is significantly harder to compile a program written in Spark, but when compiled, the chances of mistakes are much less than other languages. Note that the Spark tool contains a number of other checkers that conventional compiler do not have i.e. data flow analysis tools, static checking tools, etc.

It is suggested by using languages such as Spark, it is possible to avoid errors being introduced in the first place. Also the precision of the language allows programmers

\(^{1}\)www.sparkada.com designed by Altran Praxis
to reason logically about the program.

Note that other technologies such as Event-B suggest that code should be generated automatically. Which means, instead of manual programming the user should start with an abstract design and refine (explained in 3.1.1) it to a level, very close to source code and then automatically generate the code. This idea could also lead to a better program, by starting to construct the program properly from the start.

2.1.2 Source-Code Verification

There are several types of technologies that could be useful for verifying software. Some of these technologies are mentioned here. The aim of these tools is to verify software but to different levels of rigor.

- Discovering likely generic errors. Tools such as data flow analysers and statistical analysers which detect important paths in the code [20]. These tools check the paths that the program is likely to take and try to identify common errors. They are practical tools for automated checkings but they do have their limitations in large systems.

- Static verification tools such as ESC/Java\textsuperscript{2} check types, array boundaries, etc. For instance in Java, the problems that can ESC/Java\textsuperscript{2} find, are normally found at run-time. Note that static checkers are neither complete nor sound. Thus they do not prove the correctness of software. Note that the SPARK tool can also be categorised as a complex static checker as it uses annotations, post and pre-conditions, data flow analysis, etc to verify the written software.

- Run-time assertion checkings can also be done using tools such as JML [20], where the user inserts assertions at different parts of the code and the tool checks if they hold, during run-time.

- Model checking tools such as Spin [8], are used to establish behavioural properties of complex systems. Then automatically check, if an abstract finite-state transition of a software model, satisfies the given properties [11]. Such tools are mainly used for checking concurrent systems [11] as they can provide counter examples when a model fails, which is quite helpful for concurrent systems. Note that NASA uses model-checkers in some of their software projects. This can be a good evidence of practicality of model-checkers in industrial applications [13].

The tools mentioned here are some of the more commonly known tools. However, there exist a great deal more that are not discussed here which can be useful as well.

2.2 Z Notation

In this section a brief explanation of Z [19] is given and the translation between Z and Event-B is discussed. Note that this explanation of Z only contains the parts of Z that are related to the modelling process of this project.

2.2.1 Z Basics

In Z the main component is a schema. A schema is made of a number of other components and/or constraints about the state of the model.
An example of a schema is shown in figure [2.1]. We encountered two types of schemas:

- schemas that declared state together with invariants and constraints on the states and,
- schemas that introduced operations. Note in Z post-conditions do not have to be overstated i.e. state invariants and post-conditions on other variables can be used to derive post-conditions on relevant variables. For instance, if for invariant \( x > y \) and, \( y' = 1 \) is the post-condition, then we can derive that \( x' > 1 \).

Note that it is possible to nest a number of schemas in other schemas to avoid duplication. For example a general "certificate" schema can be nested in a "privilege certificate" schema i.e a privilege certificate contains all that a certificate does and also it has some extra functionalities. What is meant by, the privilege certificate can have general certificates nested in it, is that in the declaration part of the privilege certificate schema, the general certificate is mentioned.

Colons (:) are used to show the type of each component. For example, in order to introduce a variable called ID that has type \( N \), the following statement should be placed in the declarations section

\[
ID : N
\]

and if 0 is not an ID, in the constraints section the following statement needs to be added.

\[
ID > 0
\]

In Z if a schema contains optional variables it is specified by the keyword \textit{optional} before specifying the type. The optional keyword means that the schema may or may not contain the particular variable i.e. if a variable is not mentioned to be optional, it must be present always. For example, in order to say that an "authorisation key" is optional the following statement should be added to the schema:

\[
AuthKey : \text{optional} \quad KEY
\]

### 2.3 Tokeneer

It was decided to use the Tokeneer project as the main case study of this project for the following reasons:

- The Tokeneer project is a real world project which was a successful UK grand challenge which was undertaken by some of the experts in the field.

- The specification and the design of the Tokeneer project is available publicly and is free to be used. The specification document of Tokeneer is the formal requirements document that is written using the Z notation [19].
• Having a model of the system available makes it easier to understand the system. This allows us to focus more on finding patterns and not be purely focused on the modelling process.

Tokeneer is a project that was offered to Praxis High Integrity Systems by the National Security Agency (NSA) in the United States to redevelop an Identification Station according to their software quality standards. Informally, this ID station is responsible for reading a fingerprint and, based on a number of protocols and checks, ensure that the person trying to access an enclave is indeed permitted to enter. Using the protocol, the ID station communicates with a number of external components to perform its analysis and if the analysis result is correct the user will be granted entry.

As shown in figure [2.2] there are four external components that the ID station is connected to which are:

- Fingerprint reader
- Smart-card reader
- Door
- Visual display

![Figure 2.2: The ID Station](image)

The basic operations that are required from the ID station are described below in a very general manner. Each operation is explained to clarify its purpose as well. Note that these operations do not specify the sequence of how these operations should occur.
1. User is given initial access to Enclave.
2. User is not given initial access to Enclave.
3. User is given permission to repeat access to Enclave.
4. ID Station is started and enrolled with the information provided by the Enrolment Station. A floppy disk is used to input the initialisation data from the Enrolment Station. This operation is required so the ID Station can identify users.
5. ID Station is started and the enrolment details are already acquired. This means that a person has already started the ID station and the station has the information required for processing user requests.
6. ID Station is closed. An authorised user, i.e. the administrator, can shut down the station.
7. Security Officer changes the configuration of the ID Station. The security officer is given the privilege to change or update the configuration data that is already in the ID Station with a new set of data using a floppy disk.
8. Audit log is saved. An auditor can archive the audit log off of the system onto a floppy disk and then clear the audit log on the ID station.
9. Guard manually unlocks the door. A guard overrides the latching process and asks for the door to be unlocked manually to allow the entry of a person (enrolled or not does not matter).
10. Administrator logs on. Administrator can log on by providing their Token in the Token reader. The Token simply means a card.
11. Administrator logs off. No further explanation is required.

The requirements that are described above are the main high level operations that Tokeneer performs. Note that there is a great deal of complexity happening in the background as the communications between the components are significantly more involved. There are success and failure scenarios for each operation that are not relevant at this stage to be explained in detail.

2.3.1 Tokeneer In Z

The critical parts of the Tokeneer model are described in this section using the terminology that is used in the Tokeneer specification document. This document is available online from [6]. Note that only the parts of the system that are essential in understanding this project are described.

Praxis Documentation

The structure that Praxis has used in documenting this project is as follows:

- an informal document explaining the project and requirements using English. This document is called the "requirements" document.
- Informal requirements were modelled in Z as specification. This document is called the "formal specification" document.
• The specification was later refined and extended in a document called the "design document". In this document, operations are extended and also more invariants are added to the system.

In this project our focus is to use the specification document to capture the internal consistency of the system i.e. the core invariants and states. The reason for this choice is that we require a large number of proof obligations to increase the chance of finding patterns.

2.3.2 Specification Document

This document is the formal specification, written using the Z notation. The document specifies the behaviour of the fundamental parts of the Token ID Station. This document specifies

• Types
• Invariants
• Operations.

The operations that are covered in this document are

• Authentication and entry into the enclave
• Enrolment of token IDs
• Admin Operations
• Updating
• Configuration operations
• Door/Alarm operations.

In general, the specification is focused more on capturing the system as a whole and recording all invariants.

Certificates

One of the crucial parts of the Tokeneer model are the certificates. There are different types of certificates that are used for verification purposes of each read token (described later). The basic content that all certificates have are:

• Certificate ID,
• Validity period: the period that each certificate is valid for,
• An optional public key of the user who validated the certificate.

The certificates used in Tokeneer all contain the above data and also some specific data. The figure 2.3 depicts the structure that certificates are related to each other. The types of certificates are:

• ID Certificate:
  - contains the user’s name
and the user’s public key (used for authentication)

- Attribute Certificate: Contains the token ID
- Privilege certificate is a subclass of the attribute certificates with the extra data:
  - A level of security clearance,
  - Privileges that the user has in using the system i.e. can the user create tokens or change the state of the system in some other manner.
- Authorisation Certificate (this certificate is optional): is the same as the privilege certificate (this certificate is used for admin operations).

Figure 2.3: Certificates Structure

**Tokens**

Another important entity of Tokeneer is the token. Tokens are the data that is read from an inserted smart card. When a user inserts a smart-card into the reader, the system reads the card and calls the read data a "token". Each token is constructed from the following:

- ID: each token has a unique ID,
- and four different types of certificates:
  - ID Certificate
  - Privilege Certificate
  - I & A Certificate (Identification and Authentication Certificates)
  - Authorisation Certificate (this certificate is optional).

Tokens are used to store all the required information of a user. So when a smart-card is read, all the essential parts for verifying the identity of the user is provided. Note that for the verification process to be completed, it is necessary to read the user’s fingerprint as well.
Time

The concept of time is modelled in Z as a non-continuous entity. When time is used in Tokeneer, it is treated as a natural number. The reason that time is defined in this system is because to calculate the amount of time e.g. the door can be open before being closed or the amount of time that the alarm should be active before being set to timed-out.

Fingerprints

As explained before, it is important to identify the user and verify his or her identity. In order to increase the security even more, after the token is read and the important information about the user is available, the fingerprint check has to be performed. If the user passes the fingerprint check, it means that the user is safe and so the enclave can be opened.

Admin

The Tokeneer ID station has administrators that have higher security privileges than normal users. Admins can

- add new certificates,
- authorise users,
- manually open the door,
- set users different privileges etc.

The difference between administrators and normal users are mainly in the operations they are entitled to use. Their smart-cards and their fingerprints are read as a normal user. However, administrators have a larger range of operation compared to normal users. They can use all operations available to normal users plus a number of additional operations, such as the ones stated above, to complete their tasks.

2.4 Reasoned Modelling Patterns

The kind of patterns that we have found in this project are described in chapter 5. The main type of patterns that were found are "reasoned modelling pattern". These patterns are heuristics that can guide the modelling and reasoning processes [13]. The purpose of these patterns is, in many cases, to help the user make decisions for changes to the model in order to discharge one or more proof obligations due to a proof failure, or simplify the proof obligation so it becomes easier to prove.

2.5 Absolute Block Signalling

In railway systems it is possible for two or more trains that are going on the same direction to use a single track. However there are situations that the railway designers must design to prevent trains from crashing i.e. one train crashing to the rear of another train. The British railway system designed a system called the Absolute Block Signalling which prevents trains from clashing into one another. The essence of the ABS system is to divide the track into linked block sections and by using signals between
each section restrict the entrance of trains to that block section. The block sections allow us to evaluate whether a train exists in a certain part of the track before letting any other train enter that part of the track. The information about the ABS system is extracted from [12]. This system is described in the next chapter in more detail and is modelled using Event-B.

2.6 Summary

In this chapter we described some of the more famous formal methods to demonstrate that formal methods are not just theorem proving approaches. Also we introduced the Tokeneer system and explained the basics of Z, required to understand the Tokeneer's specification document. Reasoned modelling patterns and the ABS model were also discussed as they are required to be understood for the following chapters.
Chapter 3

Event-B & Rodin

Here, Event-B is introduced to the extent of understanding this project. The Event-B language is described in the next section by providing a relatively simple case study. Also the Rodin tool is described to demonstrate how the tool-set can be used. Before starting explaining Event-B a number of concepts are introduced.

3.1 Event-B

Event-B is an evolved version of the B method [1]. Event-B language is for modelling and reasoning about complex and discrete systems [2]. The term proof obligation is mentioned in this section regularly. What is meant by a proof obligation is, a theorem that needs to be proved in order to verify the correctness of the model.

Event-B is a formal method with first-order logic and set theory as underlying mathematical notation. There is a good tool support for Event-B through an Eclipse plug-in called Rodin. Rodin allows us to write the specifications and check their correctness (explained later in section 3.3). When modelling with Event-B most of the proofs are proved automatically by Rodin, but there are certain cases where human interaction is necessary to complete the task of proving [3]. The proof obligation generator, generates a number of proof obligations some of which will be proved automatically by Rodin and some will need human intervention to be proved. In many cases the number of proof obligations can be quite large depending on the size of the project. Thus more human interaction is necessary and consequently more time is required to complete the modelling task. However, experiments indicate many cases in which based on experts guidance or personal experience, some proof obligations can be discharged following the same line of reasoning.

The use of models (or blue-prints) to speculate system properties and future capabilities is not a very common practice in the computer industry. Unlike other practices, such as mechanical or structural engineering where blue-prints are used to speculate the future system, such practices are rare in the computer science. In cases where there are blue prints for defining system more often the model is very close to executable code which is not the purpose of designing blue-prints. For example a blue-print of an engine does not have engine features but in computer science blue-prints such as UML have qualities in common with executable code. This relationship between models and executable code gives the impression to the engineers that modelling is another high-level programming language. However, Event-B promotes an idea called refinement (explained in 3.1.1) that a user should start from a very abstract model and, refine it to a better model. This could hide some of the complexities, that had to be dealt
with, in programming at an early stage to a later stage where the user understands the system more accurately.

When discussing modelling in Event-B, we are concerned about discrete models. A discrete model is made of a state with a number of variables and constants that represent some an abstraction of the real system that is being modelled [5]. Apart from the state there are operations that can take place under some conditions which can modify the state. These transitions are called events. Each event is made of guards (conditions that must hold in order to allow actions to occur) and actions (the way that variables can be modified as the result of the execution of the event).

In order to be able to reason about the model, we need a certain number of invariants, which are conditions on variables that must always hold.

3.1.1 Refinement

Having a reasonable level of abstraction, that can take most of the facts about the real system into account, is a difficult task. In order to solve this problem, it is possible to design a model with a quite a high level of abstraction and as the design evolves, add more details to the model to reduce the level of abstraction. This concept is called refinement which is used in Event-B modelling. The analogy is that the deeper one goes into the design the more details should be revealed rather than providing all details at first. The advantage of refinement is that it makes designing a complex system relatively easier to comprehend [5]. The most important point about the idea of refinement is, by using refinement, instead of focusing on how to solve the problem, the user is directed to focus on what the system is. Thus the user understands the system better before starting to expand the system. Also it results in less complex proof obligations as the complexity is handled by abstraction [4]. In the chapter 6 there is a discussion about the use of refinement and the challenges that it can cause.

3.1.2 Decomposition

By just applying refinement alone the problem of complexity cannot be solved. Because when a model is large, refining it over and over results in a large model which would be very difficult to work with as a whole. So there is a need for a method to divide the model into smaller pieces in a systematic way. This method is called decomposition [5]. The result of performing decomposition is that the complexity of the model will be reduced and allow the designers to concentrate on smaller parts first before modelling the system as a whole.

3.1.3 Generic Development

The defined model using refinement and decomposition can be defined by using carrier sets (a set containing elements of a particular structure as opposed to a relation) and constants. This generic model can then be developed with some mathematical theory such as set or group theory.

3.1.4 Modelling in Event-B

A model has four parts which include:

- A name
• A set of state variables $v$
• A set of named invariants $I(v)$
• A set of events

Events are made of three parts which include:

• A name
• A list of guards $G(v)$. Guards are the required preconditions that must hold in order to allow the event to occur. They state the required states that the system must be in, in order to continue its operation.
• An action $S(v)$. In order to be able to describe a transition it is necessary to be able to replace a variable with a number of set-theoretic expressions. This is achieved through substitutions $E(v)$. For instance, a deterministic substitution for $x$ is $x := E(v)$.

Furthermore Event-B provides contexts (contain the static parts of the model) and machines (contain the dynamic parts of the model) which are concepts important to understand to be able to create formal discrete models.

Machines is where events and variables are defined. Each machine can see a number of contexts and use the defined constants and axioms in the context. The differences between a model (or machine) and a context are:

• A model describes variables whereas a context describes carrier sets
• A model describes named invariants whereas a context describes constants
• A model describes events whereas a context describes named properties.

Each context has four elements, as follows:

• A name
• A list of distinct carrier sets, denoted by $s$
• A list of distinct constants, denoted by $c$
• A list of specific properties, denoted by $P(s, c)$

In order to explain Event-B and demonstrate how it can be used, an example is provided in the next section 3.2 which describes a relatively simple system that is modelled in Event-B. Using the example we will illustrate Event-B.

### 3.1.5 Event-B Proof Obligations

Different types of proof obligations exist in Event-B. This helps users to understand why each proof obligation is generated. Also, because of the way proof obligations are named, users can realise on what part of the model to focus on which is related to the particular proof obligation.

Some of the types of proof obligations available in Event-B are described in this section. Note that the structure of a proof obligation name in Event-B is cause/proof obligation type where cause could be an event, invariant or axiom and the proof obligation type is one of the predefined proof obligation types in Event-B.

1Sometimes called models
• **Well-Definedness (WD)**: partial functions are used in Event-B quite frequently. Using partial functions could result in reasoning about badly-defined expressions in proofs which can be difficult and tedious to work with [4]. Therefore it is necessary to prove that partial functions are applied only to arguments in their own domain. So WD proof obligations are ensure that partial functions are never applied to arguments outside their set domain.

• **Feasibility of non-deterministic events (FIS)**: Event-B requires actions to be feasible when their guard (preconditions) are true [4]. This means that an action must yield success if its preconditions hold. Thus this proof obligations ensures of that.

• **Invariant Preservation (INV)**: ensures that the invariants hold over events. Meaning that it must be proved that invariants are always valid even after actions are applied.

Also there are other types of proof obligations for *guards, witnesses* and *theorems* (derived axioms) as well, but are not discussed in here as they are not necessary for understanding this project.

### 3.2 ABS Model

The ABS system is described in this section with its Event-B models to illustrate how systems can be designed in Event-B. The purpose of this case study is to introduce the concepts that are important in completing this project.

#### 3.2.1 Definitions

The ABS system consists of the following components:

• **The Signal Box**: consists of a Block Instrument and a Block Bell. The operator has control over the signal box and sends signals using the available instruments to communicate with the other signal boxes.

• **The Block Instrument**: is the device used to set the state of the block section. The states that can be set using the block instrument are:
  - Line blocked
  - Line clear
  - Train on line

• **The Block Bell**: is used to send different types of signals to the neighbouring signal boxes to inform them about the current state of the block section. The signals are described below:
  - Call attention (1 beat)
  - is line clear for a train (4 beat)
  - train entering section (2 beat)
  - train out of section (3 beat)

• **The Track Side Signal**: is similar to a traffic light. It has states danger and clear which will respectively, stop or allow trains to enter the section.
• **The Operator:** is the agent who controls the signal box. It has two modes, *forwarding* or *accepting*.

Figure 3.1 depicts an example implementation of an ABS track and figure 3.2 represents two signal boxes with their internal components that are linked together.

![Figure 3.1: An ABS Block Section with its components. ©Andrew Ireland](image)

### 3.2.2 The Protocol

Suppose that we have two signal boxes A and B and a block section called AB. In order to allow a train to pass from signal box A and enter block section AB and then reach the signal box B, we require a protocol which specifies how this process can take place. Below this protocol is described.

1. Operator in A should be in forwarding mode and B should be in accepting mode.
2. A signals B call attention.
4. A signals B is line clear for a train.
5. If line is clear i.e. no train is in that block section, B signals A is line clear for a train. B sets the block instrument state to line clear.
6. A hears and sees line is clear. A and B set their track side signals to clear.
7. A signals B train entering.
8. B signals A train entering. Set block instrument to train on line.
9. A sets track side signal to danger.
10. Train exits AB. B sets its track side signal to danger. Train now is in BC (i.e. the next section).
12. A signals B call attention.


14. B sets its block instrument to line blocked. While AB is blocked no more trains can be accepted by B.

15. The same process now should be undertaken for BC.

Note that because of the sequential nature of the protocol and the complexities involved in designing sequential properties in Event-B, it is not modelled in the following models. Sequentiality can only be achieved in Event-B through ghost variables\(^2\). Before starting to model the ABS system, the sequential nature of the protocol was not considered. Therefore, it was decided to model this property, however, due to time constraints and unnecessary complexities that it could bring to this brief introduction of Event-B, it was ignored.

### 3.2.3 ABS Event-B Model

The models provided below describe the main operations of the ABS system. The following are the components of the model and are explained in detail later.

- **CLine**: Context that contains the set of signal box states.

- **MLine**: Machine which describes operations for events from one signal box state to another. For example from state *line clear* to state *line blocked*. It refines CLine.

- **CSection**: This context describes block sections.

\(^2\)Variables that are not related to the actual model and they are only related to some specific event.
- **CTrackSideSignal**: is the context which describes a track side signal.
- **CTrain**: This context describes the train. The train only has an ID and it is described in this context as a set.
- **MTrain**: This machine is responsible for introducing how to move a train from one block section to another. It sees:
  - CLine
  - CTrain
  - CSection
and refines MLine.
- **MSignal**: is responsible for modelling state transitions of the track side signals. It sees
  - CLine
  - CTrain
  - CSection
  - CTrackSideSignal
and refines MTrain.

**CLine**
The states that each line can have are modelled as a set as shown below.

```plaintext
CONTEXT  CLine
SETS     LState
AXIOMS   lsa : LState \neq \emptyset
END
```
The only requirement of the states that is constant is that the set is non-empty.
Each section is modelled as below:

**CSection**

```plaintext
CONTEXT  CSection
SETS     Section
CONSTANTS back
         front
         isOccupied
AXIOMS   ax1 : Section \neq \emptyset
         backax : back \in Section \rightarrow Section
         frontax : front \in Section \rightarrow Section
         occax : isOccupied \in Section \rightarrow Section
END
```
The constants back, front and isOccupied are provided so they can be used in the axioms. The axiom front provides the next section and back provides the rear section from where the signal box is. The axiom isOccupied specifies whether a block is occupied or not. Note that \( S \rightarrow T \) is a bijection which means that it is a one-to-one relation. For instance, the axiom backax is a one-to-one relation from one section to another. This is because there is only one section behind another section. Every train is modelled as follows:

CTrain

CONTEXT CTrain
SETS
  TrainId
AXIOMS
  axtrainid: TrainId \neq \emptyset
END

The only requirement of the TrainID is not to be empty. The other modelled contexts are similar to the ones provided above. Description of the MLine machine for CLine constant is provided below (not all events are provided below because of their similarity):

MLine

MACHINE MLine
SEES CLine
VARIABLES
  line_blocked
  line_clear
  train_on_line
INVARIANTS
  inv1: line_blocked \subseteq LState
  inv2: line_clear \subseteq LState
  inv3: train_on_line \subseteq LState
  inv4: partition(LState, line_blocked, line_clear, train_on_line)
EVENTS
  Initialisation
  begin
    act1: line_blocked := \emptyset
    act2: line_clear := LState
    act3: train_on_line := \emptyset
  end
  Event line_blocked_to_clear \equiv
  any s
  where
    grd1: s \in line_blocked
  then
    act1: line_blocked := line_blocked \setminus \{s\}
    act2: line_clear := line_clear \cup \{s\}
  end
END
The invariants provided above are for ensuring that line_block, etc are of type LState. The function partition means that the shown states are part of the LState set. The action line_blocked_to_clear is provided to show how a line from blocked can go to clear. As shown, subtracting set line_blocked from the current state (set s) can be take it out of this set and add it to the clear set.

MTrain is responsible to refine MLine and add the function for moving trains to the model.

MTrain

MACHINE MTrain
REFINES MLine
SEES CLine, CTrain, CSection
VARIABLES
  line_blocked
  line_clear
  train_on_line
  train_state
  occupied_section
  next_section
INVARIANTS
  inv1: train_state ∈ TrainId → Section
  inv4: partition(Section, occupied_section, next_section)
EVENTS
Initialisation
  extended
  begin
    act1: line_blocked := ∅
    act2: line_clear := LState
    act3: train_on_line := ∅
    tact: train_state :=∈ TrainId → Section
    ne: next_section := Section
    cu: occupied_section := ∅
  end
Event move_train ≡
  any
    i
    sec
  where
    grd1: i ∈ TrainId
    grd2: sec = train_state(i)
    grd3: sec ∉ occupied_section
    grd4: front(sec) ∉ occupied_section
  then
    act1:: train_state(i) := front(sec)
    act2:: occupied_section := occupied_section ∪ {front(sec)}
  end
END

As shown the moving process requires the current section to be the section occupied and the next section should be checked to be free. If the conditions hold, the train can advance to the next section and the current section should be set to free. The next
section must be set to occupied.

MSignal refines MTrain to add the signalling capabilities to the model.

MSignal

MACHINE MSignal
REFINES MTrain
SEES CLine, CTrain, CTrackSideSignal, CSection

INVARIANTS
invdc : partition(TSSStates, danger_signal, clear_signal)

EVENTS
Initialisation
extended
begin
act1 : line_blocked := ∅
act2 : line_clear := LState
act3 : train_on_line := ∅
tact : train_state ∈ TrainId → Section
ne : next_section := Section
cu : occupied_section := ∅
dact1 : danger_signal := TSSStates
cact2 : clear_signal := ∅
end
Event signal_to_clear ⪰
any
s
where
grd1 : s ∈ danger_signal
then
act1 : danger_signal := danger_signal \ {s}
ac2 : clear_signal := clear_signal ∪ {s}
end
Event signal_to_danger ⪰
any
s
where
grd1 : s ∈ clear_signal
then
act1 : danger_signal := danger_signal ∪ {s}
ac2 : clear_signal := clear_signal \ {s}
end

END

This refinement adds the capabilities of setting the track side signals to either danger or clear as the operation is similar to previous events, it is not explained in any further.

3.2.4 ABS Proof Obligations

The ABS model was a very simple model that was just used to demonstrate the use of Event-B. Because of its simplicity, all the proof obligations were discharged automatically by the provers. The types of proof obligations (explained in 3.1.5) that were generated are:
• Five well-definedness (WD) proof obligations.
• A feasibility of non-deterministic event (FIS) type proof obligation. This proof obligation was generated because of
  \[ \text{TrainId} \rightarrow \text{Section} \neq \emptyset \]
• Five proof obligations of type INV for the invariants of the model which were also automatically discharged.

3.3 Rodin

In this section, the use of Rodin is described. The extent of this tutorial of Rodin is to the level necessary for being able to understand the concepts of this project. Rodin is an Eclipse based plug-in which means it is a multi-platform tool. It has two different perspective, one perspective is the modelling perspective where the user creates the model and the other is proof perspective which is used for reasoning about the model.

3.3.1 Basics

As explained in §3.1.4, machines and contexts are the main part of modelling in Event-B. When using the standard version of Rodin, the user has to create the model using a form that Rodin provides. The problem with these kinds of forms to be the primary way of creating a model is the amount of time that it takes to point, click and type. In order to avoid this challenge, there is a text editor add-on that is available in Rodin which can be installed to allow the user to type the model with no interaction with any forms. This text editor (Camille) was used in this project and therefore the use of forms is not discussed here.

3.3.2 Modelling Perspective

The modelling perspective of Rodin is shown in figure 3.3. In this perspective, the user creates the model and can create machines or contexts files. The modelling perspective contains the following:

• **Modelling Editor**: the location where the actual model is generated.
• **Project Explorer**: is where the projects are created, accessed and modified structurally.
• **Symbol section**: provides all the necessary symbols that are used in Event-B.
• **Perspective Changes**: as shown in figure 3.3.
• **Error Display**: errors and problems with the model are shown here.

3.3.3 Proof Perspective

The way that a user can perform interactive theorem proving in Rodin is by:

• Opening an undischarged proof obligation (note that the perspective should be changed to proof perspective, this is marked in figure 3.3).
Using the goal and the hypotheses, trying to find a method to prove the proof obligation.

In figure 3.4 Rodin’s proof perspective is depicted. In the figure each segment of the interface is named to show the meaning of each part.

- **Hypotheses Section**: is the part where all the hypotheses are gathered and saved.

- **Proof Obligations**: is the part where proof obligations, discharged or undischarged, are listed.

- **Goal**: is where the goal that is being proved is located.

- **Proof Control**: is where the user can interact with the provers. This section is described in more detail below.

- **Proof Tree**: is the place where the steps that are taken towards proving a particular proof obligation are saved.

Figure 3.5 demonstrates the proof control panel of Rodin’s proof perspective. Each important button is numbered and its functionality is described below.

1. The types of provers available to use. Using its drop down menu it is possible to choose if all the hypotheses should be included.

2. Sets the proof obligation to "reviewed", which means: assume it is proved for now. This is used to postpone the proof of some proof obligations to a later stage. By selecting a proof obligation to be reviewed, the provers will no longer attempt to prove or check that particular proof obligation.

3. For running a number of other external provers that are not part of the general provers.
4. Acronym for Add Hypothesis. By entering a hypothesis in the text box provided, it is possible to add the hypothesis to the list of all hypotheses.

5. Used for rerunning all the automatic provers.

6. Searches for the hypothesis that is written in the text box. There are times that it might be necessary to search for a particular hypothesis. Using this button can perform a search operation for that hypothesis and if it exists, it will be shown to the user.

7. The text box where the user can interact with the provers.

Note the rest of the buttons that are not described will not be necessary for this project.

3.4 Summary

In this chapter we introduced Event-B and Rodin by using the ABS system as an example. With the background knowledge provided here, the rest of the project can be understood. Rodin and Event-B are explained to the extent of completing this project.
as they are not the primary aim of the project. The most important parts of the ABS model are discussed in this chapter along with the required Event-B code. In the next chapter we use the concepts introduced to explain our Tokeneer model.
Chapter 4

Modelling Tokeneer

This chapter describes the details of how the modelling process of this project was carried out. Also the important aspects of the model are described with corresponding examples.

4.1 Introduction

As stated before (in §2.3.1) the Tokeneer project provides a number of documents that describe the system. We have chosen to use the specification document. Also as stated before the specification of Tokeneer is written in Z and the document is clear and understandable. The main part of the modelling process is converting Z to Event-B.

Another important point about modelling this system is that, it was decided to focus mainly on the core functionalities of the system, the parts which are directly related to the ID station. The core operations are those which in some way contribute to the ultimate goal of unlocking the door and allowing the user to enter.

4.1.1 Modelling Strategy

The approach for the modelling process of this project was as follows:

1. Understanding the "requirements document" which explains the system using use cases and diagrams to understand the whole system, informally.

2. Learning Z using the documentation provided by the Tokeneer project and figuring out how to translate that to Event-B notation.

3. Starting to understand the basic types and elements involved in the lowest level of the system. For instance, discovering the structure of each certificate, token and physical elements such as door or alarm.

4. Starting the model by constructing the basic, most important types such as certificates. Having these types already modelled before working on any invariants or operations makes the modelling process easier because the focus can be directed to operations or invariants that have to deal with the most important parts of the system.

5. Initially only focusing on modelling as many invariants as possible to increase the chance of generating more proof obligations as the invariants usually generate proof obligations. Also, we wanted to capture the state and invariants before
adding operations. One of the initial concerns in this project was lack of proof obligations which could risk the chance of finding patterns.

6. Modelling operations related to unlocking and allowing entry.

7. Modifications based on the undischarged proof obligations.

The strategy above, was not our starting strategy. Because of a number of issues and challenges that occurred during the modelling process, we had to change our approach. In section 4.1.2 we discuss the major changes made to the strategy of this project.

4.1.2 Strategy Modifications

When modelling was started, the strategy was not quite as specified in 4.1.1. At the start it was assumed that starting to model operations first and whenever necessary adding the required segments could be a better approach. However, this proved to be not very effective in generating proof obligations because the number of modelled invariants that were related to the operations was not large enough. Tokeneer specification is a state based specification rather than an operation based. So it was necessary to model most basic types and then model the invariants related to those types. Therefore in the middle of the modelling process, it was decided to change the approach and model the states.

Another change to the modelling strategy was that we decided not to "slice" the Tokeneer project and choose segments of it. Because of the dependencies between the variables, it was much easier to capture most important types and then add the invariants to the model. As far as the main invariants are concerned, most of them are captured in this model.

4.1.3 Difficulties In Modelling

When the modelling process started, almost all the challenges described in section 6.1 had to be dealt with. The most important difficulty with the modelling was making decisions about how to model the system. Note that the task of modelling Tokeneer was not purely translating Z to Event-B because Event-B is not exactly the same as Z.

Start of the modelling process was one of the difficult phases of the project. The questions and challenges presented in 6.2 were the main barrier in dealing with the modelling process. The most important point about these challenges was that it was not clear whether what was being done, was right or wrong. This dilemma caused the modelling process to take longer that expected. It was not clear whether the decisions made were right or wrong or whether the quality of the model was acceptable. In fact, there were numerous cases where the model had to be changed because of wrong judgements i.e. decisions about the way that refinement was being done were mostly problematic and had to be changed.

4.1.4 Using Refinement

Our first decision was to use the refinement feature available in Rodin for the modelling task but because of the challenges that were faced (caused by refinement), we decided not to use it. The challenges that exist with refinement are discussed in greater detail in section 6.2.
First Approach

At the start of the modelling process refinement was applied, as it was thought that it could simplify the modelling process. However, it was a major challenge to decide how to use refinement in the model because, as explained before, the starting modelling approach was to model operations first but it was changed to model states first. When modelling states, the focus is the entire system and the states that it can have. When modelling a system in a state based fashion, using refinement is quite difficult as it is not clear how to separate states. The system that is being modelled is a single model and its states cannot be separated.

Second Approach

Because of the challenges encountered in using refinement and the decision to model the system in a state based fashion, it was decided not to use refinement. So the system was modelled in one "machine" and one "context". Avoiding refinement helped in modelling because the critical decisions that need to be made while refining, were non-existent. Later when operations were added, we did not use refinement.

4.2 Tokeneer Model

The Tokeneer system was described in 2.3. In this section the Event-B model of Tokeneer and the most important aspects of the model are discussed. The parts of the system that are modelled in this experiment are:

- Tokens
- Token ID enrolments
- Certificates
- Fingerprints
- Door
- Latch
- Alarm
- Admin User Type

The specific operations that are modelled are the following:

- **Reading user token**: This operation is responsible for reading an inserted token.
- **Tearing token**: In case the token is invalid, using this operation allows to discard it.
- **Reading with no bio test**: If checking the fingerprint is not required, this operation will be used for updating the state.
- **Reading fingerprint**: Scanning the user fingerprint from the fingerprint scanner.
- **Timing the validation of a fingerprint**: Setting (calculating) the validation period of a fingerprint.
• Validating fingerprint: Checking the fingerprint is valid. But in this model checking the actual fingerprint is the responsibility of the fingerprint scanner. This operation just updates the state of the system to what it has to be after reading a fingerprint.

• Writing the new user token: Recording the token. Note that the token in this model is referred to as the token ID.

• Allowing entry, before unlocking: Changing the state of the system before unlocking the door.

• Unlocking the door: State changes for unlocking.

• Locking the door: State changes for locking.

These are the core functionalities of the Tokeneer system although there is a great deal more that can be added but due to time constraints, it was decided not to extend the model since enough proof obligations were generated.

4.3 Conversion of Z to Event-B

This section discusses the strategies, methods and important points about how we converted Z notation to Event-B notation. In §A and §B the Event-B model of this project is provided.

4.3.1 Carrier-sets & Schemas

In Event-B a schema can be viewed as a carrier set or a type, depending on state changes. Based on the style of modelling and the required properties, it is possible to view a schema as a type and use functions to relate the type to its variables. For example, in this project the type certificate (a schema originally described in the specification) is a carrier-set that contains all different types of certificates. To be able to relate certificates to their other properties such as their IDs, functions are introduced that ask for an ID and then return a particular constant or variable that is related to the original certificate's schema. The example below shows how the certificate schema (4.1) was converted to Event-B notation.

```
SETS
Certificate
CertID
time
KeyPart

VARIABLES
ID
```

Figure 4.1: Certificate Schema, Tokeneer Specification
valPeriod
isValBy

ININVARIANTS

inv1: ID ∈ certs → CertID
inv2: valPeriod ∈ CertID → ℙ(time)
inv3: isValBy ∈ ID → KeyPart

END

So as it is demonstrated above, a relatively small schema in Z, when converted to Event-B, requires some effort. Note that in order to have relations in Event-B it is necessary to introduce functions.

4.3.2 Conversion Decisions

It is important to note that this conversion from Z to Event-B requires some level of personal choice as it is possible to convert Z to Event-B in many different ways without changing the semantics. During the modelling process of this project, it was decided to follow the specification of Tokeneer and reduce personal changes to the model to avoid changing the semantics and cause unexpected problems while extending the model. For example, it is possible to represent a "token" as a carrier-set (because it is a schema) and a "valid token" as another carrier-set (as it is also a schema which has token schema nested in it). But it was decided to introduce a carrier-set for tokens and then make a valid token to be a subset of tokens.

Of course, for the above example, the chosen approach of conversion is the most relevant conversion of Z to Event-B. However, it was possible to represent valid tokens as another carrier-set which is very similar to tokens but the invariants related to it are different. This would be possible, but the problems that it causes because of the relations between tokens and valid tokens during the later stages of the project, potentially, could have been very time consuming to deal with. Thus understanding the core types and operations of the system before starting the modelling process was a crucial step towards modelling as it had a significant affect on the translation decisions of Z to Event-B.

Moreover, decisions on what should be considered constant i.e. placed in the context and, what should be considered dynamic i.e. placed in the machine, had to be made. The reason for this was that the specification document does not specify static or dynamic properties. So when a segment is modelled, if it was not modified later in the specification, it would remain as a constant. For instance, certificates are modelled as constant because in the segments that were modelled, the state of certificates do not change.

4.3.3 Optional Keyword

As explained in 2.2, an optional keyword means that a component may or may not be present. The conversion of this to Event-B is the use of a partial function to represent the relation between the particular set and another set.

4.4 The Model

In this section some chosen segments of the model are extracted and described to demonstrate how the system was modelled. All the segments of Tokeneer that are

35
described here are explained in more detail in section 2.3.

Most of the modelling, as stated before, was focused mainly on the core functionalities of the system. As the result, most of the invariants are concerned with tokens and certificates as these entities play a key role in verification of users in the ID station. Most invariants are concerned with types which are quite trivial which do not need further elaboration. However, some of the more complex invariants that are related to the segments that are discussed below, are explained to demonstrate the origins of the difficult proof obligations.

Note that the notation used in describing the modelled segments is Event-B notation. This means that, for instance, for defining types of variables, we do not use colon (:) as in Z. Instead we use the Event-B way of specifying a type which is by specifying a type invariant.

4.4.1 Time

The concept of time is an important part of this model as a number of verifications and validations require time to be checked. In this model, time is modelled as a non-continuous component of the model. It is set to \( \text{time} = \mathbb{N} \) and all the other variables that represent time are elements of the \( \text{time} \) set.

4.4.2 Tokens

In the specification document tokens are represented in a schema called token. Each token has an ID and four different types of certificates. The way that tokens are modelled is by introducing a carrier-set TokenID and creating a variable called tokenID. The ID was chosen because it is unique and it allows to define functions that can relate each tokenID to different types of certificates.

Valid Tokens

A valid token is a token that

- is of type tokenID,
- its ID certificate’s ID is equal to its privilege certificate’s ID and,
- its ID certificate’s ID is equal to its I&A certificate’s ID and,
- the tokenID that is related to its attribute certificate is equal to the token ID of the current token i.e. the valid token that is being validated.

This invariant is defined in Event-B notation below:

\[
\text{validToken} \subseteq \text{tokenID} \land (\forall x. x \in \text{tokenID} \Rightarrow \neg \text{validToken} \leftrightarrow (x \in \text{validToken} \leftrightarrow (\text{ID}(\text{tokIdC}(x)) = \text{ID}(\text{tokPrivC}(x)) \land \\
\text{ID}(\text{tokIdC}(x)) = \text{ID}(\text{tokIAC}(x)) \land \\
\text{attCertTokID}(\text{tokAttC}(x)) = x))
\]

This invariant generated a WD (well-definedness) proof obligation which is related to the way the invariant is defined. There are a number of sub goals that need to be
proved in order to discharge this proof obligation. The starting sub goal of this proof obligation is

\[ x \in \text{dom}(\text{tokIdC}) \]

Although a large amount of time was spent on proving this particular proof obligation, because of:

1. having a large number of hypotheses (mostly irrelevant) that Rodin had found, it was very time consuming and difficult to know which hypotheses were related to the goal.

2. Also, there were a number of related hypotheses that required providing a witness for some of the variables. However, knowing what the initialisations should be, was a major challenge. As a result of lack of documentation and time, proving this proof obligations had to be stopped.

The challenge with this WD and two other WD proof obligations (explained later) was that they were all very similar and the chance of discovering some kind of pattern during the proving process of these POs was quite significant. However, because of the mentioned setbacks, the process had to be postponed to a later time.

### 4.4.3 Certificates

As explained before, there are different types of certificates that are related to a particular token. The way that certificates are modelled is by having a carrier-set called \( \text{CertID} \), because all certificates are identified by their ID. This carrier-set is the parent of all certificates. All other certificates are subsets of this set.

\( \text{Certs} \) is the set of all certificates which itself is the only subset of \( \text{certID} \). This is the exact conversion of the type structure of certificates that is represented in the Tokeneer model (as depicted before in figure 2.3).

The following statements show how this structure was represented in Rodin. Note that the partition function is used to partition the sets \( \text{certs} \) and \( \text{attCert} \).

Partitioning a set \( S \), \( \text{partition}(S, x, y) \) means that \( x \) and \( y \) are distinct parts of \( S \) so the intersection between the two sets \( (x \text{ and } y) \) is empty and the union of \( x \) and \( y \) is \( S \) i.e. \( x \cup y = S \).

\[
\text{partition(\text{certs}, \text{idCert}, \text{attCert})}
\]

\[
\text{partition(\text{attCert}, \text{privCert}, \text{iandACert}, \text{authCert})}
\]

### 4.4.4 Current Token

There is a variable \( \text{status} \) which is responsible for recording the state of the system at any time. The available statuses that are available are described below:

- Quiescent
- Got User Token
- Waiting Fingerprint
- Got Fingerprint
- Waiting Update Token
• Awaiting Entry
• Waiting Remove Token Success
• Waiting Remove Token Fail

If the status is in any of the following states,
• Got Fingerprint
• Waiting for Fingerprint,
• Waiting to Update Token,
• or Waiting Entry.

Then the currentUserToken variable (that holds the current inserted user token (not admin)) should be a valid token with valid authorisations.

This invariant is provided below to demonstrate the representation of this invariant in Event-B notation.

\[
\text{status} \in \text{gotFinger, waitingFinger, waitingUpdateToken, waitingEntry} \Rightarrow \\
(\exists \text{vt}. \text{goodT}(\text{validTokenID}(\text{vt})) = \text{currentUserToken}) \\
\lor (\exists \text{twva}. \text{goodT}(\text{tokWithValidAuthID}(\text{twva})) = \text{currentUserToken})
\]

This invariant generates a proof obligation that is a WD proof obligation which its starting sub goal is

\[
\text{vt} \in \text{dom(\text{validTokenID})}
\]

and for the same reason as above it was not discharged.

4.4.5 Roles

There are three different types of administrator privileges that are

• Guard
• Security Officer
• Audit Manager.

Depending on what privilege an admin has, there are different operations that the admin can perform such as adding new certificates or changing ordinary user privileges.

In order to relate an admin token with its available privileges, the function rolePresent was introduced.

\[
\text{rolePresent} \in \text{tokenId} \rightarrow \text{AdminPrivilege}
\]

Note because not all tokenIDs have admin privileges, the function is a partial function. Another property that this function has, is:

\[
\text{rolePresent} = \emptyset \Rightarrow \text{availableOps} = \emptyset
\]

This invariant is discharged automatically and needs no further action. However there is another invariant which generates an undischarged proof obligation. This
invariant states that if there are roles available for a token, then there must exist an admin token with valid authentication that is also a good token (meaning it is still available and the system can read it).

\[
\text{rolePresent} \neq \emptyset \Rightarrow \\
(\exists x. x \in \text{tokenWithValidAuth} \land (\text{goodT}(x) = \text{currentAdminToken}))
\]

This invariant results in a proof obligation that is also a WD proof obligation with starting sub goal:

\[
x \in \text{dom(\text{goodT})}
\]

and for the same reason given in [4.4.4] it was not discharged.

4.4.6 Operations
The main focus of this project was to capture more invariants to increase the chance of PO generation, but a few core operations were also modelled to make the model more natural. The modelled operations are the following:

- Reading tokens,
- Validating tokens,
- Lock/Unlocking door,
- Allowing entry,
- Checking fingerprint,
- Setting timeouts after state changes.

These operations are some of the core functionalities of the Tokeneer model that were modelled in this system. They, generally, are the same as Z, however, in "reading token" operation an extra statement was added to the model.

Reading Token
The event below is the reading a token operation. The operation describes the following:

- If there is a token available which is of type tokenID and,
- the status does not have a token being read and,
- the enclave is either in idle mode or waiting for another admin token,
- then the token can be read and the state of system has to be changed.

**Event**  \( \text{readToken} \triangleq \\
\text{any tok} \\text{where} \\
\text{grd1 : tok} \in \text{tokenID} \\
\text{grd2 : status} \neq \text{gotUserToken} \\
\text{grd3 : enclaveStatus} \in \{\text{enclaveQuiescent, waitingRemoveAdminTokenFail}\} \\
\text{then}
The action act1 is added by the modeller’s choice. The reason for adding this statement is that the newly read token has to be added to the set of current tokens (which is a singleton set). This statement was not specified in the main Tokeneer specification because more elaboration on this operation is provided in the design document which was not the focus of this project.

Tearing Token

The operation below is responsible for removing the current token. The operation is self-explanatory, however, note that the set currentToken is set to empty after the token is torn (because the set is a singleton set). This statement (act1) is added as the result of the choice made in "reading token" event.

**Event** \( \text{tearToken} \triangleq \)

\[
\begin{align*}
\text{any} & \quad \text{tok} \\
\text{where} & \\
\quad \text{grd1} : & \quad \text{tok} \in \text{tokenID} \\
\quad \text{grd2} : & \quad \text{tok} \in \text{currentTok} \\
\quad \text{grd3} : & \quad \text{status} \in \\
& \quad \{ \text{gotUserToken, waitingUpdateToken, waitingFinger, gotFinger, waitingEntry} \} \\
\text{then} & \\
\quad \text{act1} : & \quad \text{currentTok} := \emptyset \\
\quad \text{act2} : & \quad \text{status} := \text{quiescent} \\
\quad \text{act3} : & \quad \text{userTokenPresence} := \text{absent}
\end{align*}
\]

4.5 Summary

In this chapter we demonstrated the method that was used to convert parts of Tokeneer to Event-B and discussed our strategy that was chosen to model this system. We explained how the conversion decisions were made. Some of the challenges in the modelling process were mentioned but not to a great extent, as they are discussed in 

All important segments of Tokeneer that were modelled were also explained and, interesting points about each segment were mentioned as well. Example proof obligations were mentioned to demonstrate the causes of undischarged proof obligations i.e. the kinds of invariants etc.
Chapter 5

Patterns & Guidance

5.1 Initialisation

The main pattern that was found during the modelling of the system was replacing the initialisation of variables from the empty set to a constant initialisation variable. During the modelling process, most variables that had to be initialised were initialised by the empty-set ($\emptyset$). The reason for setting the initialisation value to $\emptyset$ was, in most cases, a mistake. It was assumed that the variables that are used to represent functions or used to be a member or a subset of other sets, should be initialised to $\emptyset$ as the system is empty initially. However this was not the case in most of the initialisations. This assumption caused a proof obligation to be generated for some of the initialisations that represented a function or they were used in complex invariants. All such proof obligations required interaction with the prover where the existence of the $\emptyset$ needed to be proved. Thus in each proof obligation that the $\emptyset$ was part of the goal, the model was modified by replacing the initialisation value of variables. For variables representing functions, the initialisation was replaced by another function which used the initialisation variable. Some of the examples of such proof obligations are provided below with their corresponding replacement.

5.1.1 Examples

The goal of this proof obligation is provided below:

$$\emptyset \in initvalidid \rightarrow attCert$$

The reason that this proof obligation is generated is because

$$tokAttC := \emptyset$$

This means that the variable $tokAttC$ that represents

$$tokAttC \in tokenID \rightarrow attCert$$

is incorrectly initialised to $\emptyset$ because the assumption that the set of certificates is empty initially is incorrect. So as explained above, the strategy for correcting this issue is to create an initialisation constant that can be used in a function to fix this problem. The function that is used for solving this proof obligation is

$$tokAttC : initvalidid \rightarrow attCert$$
The sign $\in$ means that $tokAttC$ is arbitrarily chosen from the set i.e.

$$TokenID \times Certificates$$

In other words, the statement means that $tokAttC$ can be any value that is non-deterministically chosen from the total function $initvalidid \rightarrow attCert$.

The following proof obligations are other examples of this pattern.

- $\emptyset \in initvalidid \rightarrow authCert$  
  This goal was generated as a result of initialising $tokAuthC$ to $\emptyset$.

- $\emptyset \in initvalidid \rightarrow iandACert$  
  This goal was generated as a result of initialising $tokIandAC$ to $\emptyset$.

- $\emptyset \in initvalidid \rightarrow privCert$  
  This goal was generated as a result of initialising $tokPrivC$ to $\emptyset$.

All the above proof obligations were changed to an easier proof obligation by replacing the initialisations to the following:

$VariableName :\in initvalidid \rightarrow certs$

When we initialise a variable (representing a function) to such a statement, we avoid using the $\emptyset$ (if $initvalidid$ is not empty). This does not change the semantics of what is represented in Z for such functions. For example in Z a function that returns the ID of token is defined as $variable = token.ID$. This does not specify what ID or even what token. So when the conversion in Event-B is non-deterministically choosing a value for token and an ID, the semantics remain the same.

These proof obligations that are examples of variables and functions that are related to certificates. However, this pattern is not just limited to variables related to certificates. For instance the PO goals below are other examples of $\emptyset$ initialisation.

- $\emptyset \in \{initFingerprint\} \rightarrow \{badFP, noFP, initFPTry, goodF\}$

- $\emptyset \in initvalidid \rightarrow TokenTry$

In all undischarged proof obligations that the empty set was the problem, using this approach was the only remedy to make the provers automatically discharge such proof obligations. The pattern is defined below in a structured manner:

1. Checking the proof obligation.

2. If the goal of the proof obligation contains the $\emptyset$ the reason is because of the initialisation to $\emptyset$. If Rodin does not allow any rules to be applied to the goal, the fastest approach to discharge the proof obligation was to modify the model.

3. Replace the initialisation value of the variable to an undefined constant for initialisation.

4. If that variable represents a function, replace the initialisation value with another function that uses the predefined initialisation constant.

Note that the proof obligations that were related to functions were replaced by a different proof obligation that required the user to prove the initialisation function. The newly generated PO was as the result of the change easier to be proved.
5.2 Numbers

In Event-B the set of natural numbers (\( \mathbb{N} \)) is not a type and is defined in terms of the set of integers (\( \mathbb{Z} \)). Initially this was not known and a few peculiar type errors occurred during the development of the model. However, a pattern was discovered when dealing with numbers in Rodin.

As explained above, the operations for "allowing entry" and "door unlocking" required to calculate new "timeout" and "durations" for different parts of the system. All variables related to time were given the type \( \text{time} \) (which is the set of \( \mathbb{N} \)). What this means is that we assume that such variables are all in fact natural numbers. Therefore adding these natural numbers together, will certainly result in another natural number. But Rodin generated a few undischarged proof obligations which required this addition operation to be proved. The following goals are the goals of the undischarged proof obligations:

\[
\text{currentTime} + \text{latchUnlockDuration} + \text{alarmSilentDuration} \in \text{time}
\]

(this is related to \( \text{alarmTimeout} \) calculation after unlocking the door.)

\[
\text{currentTime} + \text{latchUnlockDuration} \in \text{time}
\]

(this is related to \( \text{latchTimeout} \) calculation after unlocking the door.)

\[
\text{tokenRemovalDuration} + \text{currentTime} \in \text{time}
\]

(this goal is related to \( \text{tokenRemovalTimeout} \) variable that must be updated after allowing entry.

Proving that the sum of these variables is in the set of natural numbers seems to be quite trivial but because Rodin did not list the following assumptions in the hypothesis section, discharging the proof obligations needed user interaction. In all three cases by adding the following hypotheses, the provers managed to automatically prove the POs (adding hypotheses is explained in previous sections).

- \( \text{currentTime} \in \text{time} \)
- \( \text{latchUnlockDuration} \in \text{time} \)
- \( \text{alarmSilentDuration} \in \text{time} \)
- \( \text{tokenRemovalDuration} \in \text{time} \)

Using these hypotheses can help the provers to prove that the sum of natural numbers is also a natural number. It is important to note that these variables are defined as invariants because they change during the time that the ID station is running. Thus it is not possible to add them as axioms to the "context" and they must exist in the "machine". As mentioned, this is a very trivial case of interactive proof with Rodin, but, it can be categorised as a pattern.

The reason that this pattern was found was because of lack of relevant hypotheses found by Rodin. Almost in all arithmetic operations, just by adding the relevant
hypotheses, the provers could prove the goals automatically. Therefore it is important for novice users that when a very obvious case cannot be proved by the provers, a reasonable place to check is the hypotheses section.

5.3 Summary

In this chapter we mentioned two types of patterns that were found during the modelling and reasoning phase of this project. These patterns are related to the use of Event-B and Rodin and are considered to be of type "reasoned modelling patterns". Note that, the identified patterns are mainly helpful to novice users with not a great deal of background in formal methods.
Chapter 6

Evaluation

In this chapter the challenges and issues that were discovered using formal methods (as a new user to the practical use of formal methods), are discussed. The challenges with formal methods in general and also the experience of using Rodin are the topic of this chapter. This chapter is focused mainly on the potential issues that could discourage new users from using formal methods.

Note that these challenges are not purely focused on a novice user. The reason is that in the "Verified Software: Theories, Tools and Experiments 2010 (VSTTE 10)" conference, it was mentioned by some of the leading experts in the field that there are a great deal of significant challenges that need to be addressed. The challenges such as

- Large number of tools and different ideas that are available and it is not clear how to use any of them (mentioned by Cliff Jones),
- Lack of good graphical user interfaces (mentioned by J Moore),
- Challenges with formal methods tools that new user and even developers of them face (Cliff Jones).

Also in [2], some of the problems identified in this project, are mentioned by Abrial. For instance, some of the problems that Abrial mentions are:

- Confusion between modelling and programming,
- Need for guidelines in the development process,
- Lack of good tool support,
- Difficulties in producing an accurate requirements document before starting the process [2].

So it is important to note that some of these issues are not just because of lack of expertise. There are some genuinely significant challenges with formal methods that need to be solved in order to attract more users to this field.

6.1 Analysis of Formal Methods

In this chapter our focus is to state the problems that a non-expert user might face when starting to use formal methods. We also provide some suggested solutions to these problems.

1http://www.macs.hw.ac.uk/vstte10/Home.html
The issues that are discussed here are not meant to degrade the usefulness of formal methods or other mathematical techniques used for software development. However, the purpose of identifying these difficulties is primarily for assisting users, unfamiliar with formal methods, in making a decision on whether to use formal methods (in their current state) or to use a different approach. These issues were identified in the course of this project while modelling and attempting to prove the generated proof obligations.

It is also important to note that we are interested in making formal methods applicable to all kinds of software projects rather just safety critical systems. Because using testing for the next generation of software systems, namely concurrent systems, is not a practical approach. It is an extremely time consuming and difficult task. At this stage, in order to realise how to make formal methods more practical, it is essential to identify the difficulties that might discourage users from using formal methods.

Many of the challenges specified here are essentially, because of lack of rules-of-thumb in using formal methods. For instance, in software engineering disciplines there are numerous kinds of rules-of-thumb that are just experts' opinions and experience in dealing with issues. The idea of design patterns that was introduced by Gamma et al. [10], was primarily aimed at providing such heuristics for software engineers to deal with these challenges. Although such heuristics are not documented in formal methods, in discussions with some of the leading experts of the field it was realised that such heuristics exist.

6.2 Difficulties In Modelling

In the modelling process there are decisions that need to be made about how to model the system. For example the following questions can be asked during the modelling process whose answer is not instantly obvious for inexperienced users.

- How to evaluate the quality of the model and the decisions that are made?
- How to know that the strategy in the modelling decisions is correct and will not cause the model to be changed fundamentally in later stages?
- How to make decisions regarding refinement?
- To what level of detail should the modelling be done?
- Are the generated undischarged proof obligations because of the modelling decisions and representation, the user's theorem proving skills or the prover's inability to prove the proof obligations?
- What is exactly meant by modelling and design?
- Should the model be more concerned about the overall system or very small details of how operations should take place?
- If the idea of modelling is to have a blueprint of the system, is focusing on the small details of the system at the modelling stage necessary?

Such questions and doubts that occur during the modelling process can reduce the progress of modelling significantly. In fact such questions were a major challenge during the course of this project. For a novice user it is quite difficult to know whether the chosen approach and decisions are appropriate, or likely to cause problems and
fundamental changes to the model. It seems that currently the answers to these questions are primarily based on previous experience and personal judgements. Because in the reviewed papers and published materials, the main discussion is the features and technical properties that tools and the notation itself offer. It is not clear how a novice user with no previous experience should "think" about modelling and approach it in a disciplined way to avoid any major difficulties in the course of the modelling.

As mentioned before the idea of design patterns in software engineering has a significant importance in providing guidance to software engineers on how to approach software design. Since the introduction of design patterns in software engineering discipline a large number of books and tools have been published on helping users in their design tasks. For instance in [10] a large number of rules-of-thumb are introduced by some of leading experts in software engineering. Rules such as "Stable Dependencies Principle" or "Interface-Segregation Principle" are principles that have shown to yield higher quality designs. So, if changes need to be made, the least amount of work is carried out. However, such disciplines are not common in formal modelling methods.

Note that the solution to this challenge could, potentially, be solved by introduction of patterns in formal methods. The patterns that were identified in this project, although not very high level, can provide some guidance for users on how to deal with proof obligations. For instance, the pattern related to numbers (5.2) can provide some guidance for users on dealing with proof obligations that contain numbers.

6.3 Modelling and Implementation

One of the issues that is quite difficult to handle when starting to model a project in a system such as Event-B is the line between modelling and implementation. It is quite difficult to know when to stop modelling and start the implementation process. This is a very important point that has to be quite clear to users when modelling. In [2], Abrial specifically discusses the need for a method to design blueprints of systems where we can reason about the future systems. He provides an example of a "car manufacturer blueprint", where you cannot "drive" the blueprint. But the problem in software development is that the modelling process is significantly similar to the implementation process. Users are focused on very small details of the system and in a system such as Event-B the user also specifies how events should take place to a certain extent, just like how a programmer writes a function. This similarity makes it difficult to identify the difference between programming and modelling.

Note that in Event-B when modelling a software system, it is encouraged to use refinement to a level close to source code i.e. the model is expanded enough that nearly all aspects of the system are modelled in a way that the system knows how to carry out each process and then the system will generate code from the model [2]. However, it is known that no single programming language is the right tool to be used for all kinds software development. It does not seem to be practical to expect everyone to use such a system. However if Event-B developers make it possible to generate code from a model in all sorts of languages, that is a different argument.

It is possible to argue that using "Correctness by Construction" (discussed in 2.1.1) approaches are more useful compared to source-code verification approaches because instead of a very complex model, the user at each stage has executable code. However, it depends on the problem that is being solved. Moreover, correctness by construction does not address the challenge of design and the strategies that should be followed for creating a design. In [7], Amey discusses correctness by construction and how
their tools can lead to correct software but he does not discuss any issues regarding modelling.

6.4 Mapping Specification to Implementation

Not all formal methods offer to generate code automatically from the specification. For example other methods such as VDM \cite{VDM}, require a method to map the specification to the implementation. This means that it is necessary to somehow show that the implementation of the program satisfies the documented and proved specification. However, the issue is that the method of performing such mappings are not well defined. For instance if a user prefers to write his specification in Z \cite{Z} and then prove the essential theorems in a theorem prover, the tool that is responsible for checking the implementation against the specification is non-existent. In the Tokeneer system where Z was used to model the system, was just the blueprint of the system. Praxis used the Spark tool \footnote{http://www.sparkada.com/} which follows correctness by construction approach. Thus the written specification was not taken into account during the implementation process but verified at the source-code level.

This is an indication for a need of some tool that can check whether a particular implementation follows the corresponding specification. Note that this issue is mainly about formal methods that do not offer automatic code generation from specification.

6.5 The Idea of Refinement

In Event-B the use of refinement is encouraged and is counted as one of the advantages of Event-B which is implemented in Rodin as well. What refinement means, informally, is that the modelling process should start with a very general, abstract model of the system, and then the extensions should be a "refinement" of the previous model. For instance, when modelling a train track, the first refinement of the model is a track and then later refinements add trains, signals, operators, etc.

Although this idea of refinement could potentially help to simplify problem solving and help the user to understand the system properly before thinking about what should be solved, there are a number of challenges with it. The challenge with refinement is that, for a novice user it is extremely difficult to make decisions on what should be allocated in each step of the refinement. This is a major challenge because these decisions on what should be allocated in each step of refinement directly affect the next steps of the refinement process. This means that there is a risk that if the user makes a wrong judgement during the refinement process, there is a chance that the whole model has to be changed. For instance, in Rodin the way refinement is implemented is that the user starts from a model and refines each model in a way that all refinements are chained together and are tightly dependant on each other. So a mistake in the starting abstraction could cause the whole model to be changed and the effort of modelling to be wasted.

In spite of refinement being a noble idea in problem solving techniques, the dependency that it creates in the modelling process is quite risky. One of the main software engineering disciplines is that dependency in design and modelling should be as low as possible \cite{dependency}. Because software systems are prone to change and therefore the risk that
model might be invalidated because of a change is quite high. Thus it is necessary to avoid dependency as much as possible.

It seems that refinement and modularity (decomposition in Event-B) should be considered together and not as separate ideas. It is important to incorporate modularisation into the model as much as possible to reduce the risk of the cost of changing the model when using refinement. The result of modularity and low dependency in the model has proved to be invaluable in software engineering [10]. There are numerous software engineering patterns that are created to deal with the issue of dependency.

So if it is decided that refinement should be used, the challenges that a novice user must deal with are:

- How to determine what should be allocated in each step of refinement.
- How to modularise the system properly to reduce the risk of changes to the model.

To answer these questions one major factor is previous experience and expertise in modelling and designing. But note that modularisation is not implemented in Rodin, although it is possible theoretically. The work that is currently being carried out by Maria Teresa Llano with authors of [13], is attempting to address such issues.

6.6 Measuring Quality

In order to approach designing and modelling in a disciplined way, it should be possible to judge the difference between designs and have some sort of method to judge the quality of a design. This is important because the way to improve a design is by knowing its deficiencies. For example how could a novice user know what is the difference between his design and an expert’s design. When a user cannot judge what is the difference between a good design and a bad design, it is not clear what should be improved. Of course such terms such as "good" or "bad" are relative terms and are not easy to judge a design using such terms. However, it seems that a method that could give some guidelines to the user to help him/her on what aspect of the design to focus on, could potentially lead to better designs.

One way that could help users to judge the quality of their designs and models are by prioritising the non-functional aspects of the design. For instance a ranked list such as the one provided below, could give the user some sort of guide to know what aspects of the design need more attention than the others.

- Modularity: How important is modularity?
- Scalability: How scalable should the design be?
- Abstraction: What level of abstraction is sufficient?
- Size: How large the design and the final model should be?

Deciding on the importance of the above qualities in the design process has the following advantages:

- The user has some sort of idea on how to start the modelling process because of a guideline on where the thinking process should be focused is provided.
- Also the user can judge whether his design is as good as an expert’s design or not. It makes it possible for others to also judge the design or suggest changes based on the initial prioritisation of the important aspects of the design.
Note that this is just a suggestion on how to solve this issue. However it seems that a great deal of the challenges that a non-expert user could potentially face in formal methods could be solved by some common heuristics.

6.7 Role of Experience

Many of the examples of successful projects that used formal methods were carried out by the experts in the field [20]. It seems that in practical use of formal methods the role of previous experience is a defining factor. Certainly in this project, all the difficulties encountered could be linked to lack of previous experience. However, in order to take some steps towards making the application of formal methods applicable to a wider range of software problems and not just safety critical systems, it is necessary to reduce the dependency on previous experience

- to help novice users to start the process with less overhead in decision makings
- and use the previous experience of the experts to guide their work.

Although the work that is being done by the AI4FM project (http://www.ai4fm.org/) is focused on helping the users in the theorem proving process, it is necessary to use the previous experience of the experts to a much a greater level and in different aspects of modelling and designing.

6.8 Undecidable Logic

Event-B is an undecidable process which means that it is not known whether proving a theorem would succeed. So for a novice user it is impossible to know or have some sense that the fact that a theorem is extremely difficult to prove is because of

- his theorem proving skills,
- the mathematical complexity of the theorem itself or,
- the model has some major flaw in it.

For encouraging the wide use of theorem proving techniques this could be a major drawback. However, when programming and using the traditional methods of "verification" i.e. testing, there is always this "hope" that even if a bug is found, the programmer knows that it is fixable and the program could work, eventually. Of course this does not prove anything about the correctness of the system. But the fact that the chances of being stuck in a way that the user might be stuck in proving a theorem is quite rare.

This is a major setback because in industrial applications time and cost are a major part of the project. The risk and cost of being stuck in formal methods where no progress can be made, should be somehow dealt with to attract more users to use such methods. Note that this problem is mainly for non-experienced and practical users i.e. programmers and software engineers.
6.9 Rodin Experience

During the course of this project Rodin was the only tool-set that was used. It provided a great deal of features, of which some were significantly helpful. However there were some difficulties and challenges that had to be dealt with. Here, the experience of using Rodin, during this project, is discussed. Note that some parts of Rodin remained unexplored because of the time limit of this project. For example the ProB animator and model checker that is available in Rodin was not used.

6.9.1 Advantages

A number of positive points about Rodin are discussed here to demonstrate what features it provides. The detailed explanation of Rodin and its use is provided in chapter 3.

User Interface

The interface that Rodin provides has proved to be quite an intuitive and user friendly design. Especially the Proof perspective of Rodin is very helpful as it helps the user to see the goal, hypotheses, proof obligations and the proof tree all together. The figure 6.1 shows this interface with explanation.

Camille Text Editor

There is a text editor "Camille" which is a plug-in that can be installed and used. This editor allows users to type their models as text, rather using other ways of creating the model. This editor was used in this project and it was very helpful. Note that this editor comes as an add-on which means the user needs to install it manually.

Completion Menu

Camille also provides a very helpful drop down menu to help the user to choose from a menu that has all variables and constants that exist in the model. In figure 6.2 this feature is demonstrated.
Proof Obligation Naming

One of the great advantages that Rodin, has and it is mentioned in [4], is the naming of proof obligations. All proof obligations in Rodin are named in a way that can direct the user to the exact part of the model that has caused the generation of the proof obligation. For example if a proof obligation is generated because of the type of initialisation, then Rodin names it as "INITIALIZATION/InvariantName/Type-of-PO".

6.9.2 Disadvantages

The issues and problems that were encountered during the development of the project with Rodin, are discussed in this section. Note that some of the issues that were experienced could be because of user’s lack of expertise in using Rodin.

Performance

During the modelling process Rodin’s performance was a major constraint on the speed of the modelling process. Rodin performed significantly slowly while modelling a relatively small system. The system which was running Rodin had the following specification:

- Rodin Version: 1.3.1
- Operating System: Ubuntu Linux 64bit
- Java Run-time Environment: 1.5.22 32bit architecture
- Hardware: Intel Core 2 Duo 2.4GHz & 4 Gb Memory

In this section the performance of Rodin is evaluated.

Building

When the model was still about 100 lines of Event-B code, the building process, required quite a reasonable amount of time to complete its task (note that the building process includes generation and proof of proof obligations). However, when the size of the model grew to more than 400 lines, the build took nearly 10 minutes to be completed. This means that every time the model changed and was saved, it took Rodin
10 minutes to be usable again. Although it is suggested that by marking some proof obligations as "reviewed" (i.e. tell the provers to ignore them for now), it is possible to expedite the process. However, it is a time consuming task as it is often necessary to check how the changes made to the model affect all the proof obligations. This inefficiency caused the interactive theorem proving to be quite difficult and even more time consuming which was one of the causes that the proving task of the whole model was not completed.

Proof Tree

Also using clicking on different steps of a proof tree to view how the proof was done, took about 5 seconds each. So when the user clicked on any step of a proof obligation's proof tree, it took 5 seconds for Rodin to display the content of that particular step. This was another challenge during the interactive theorem proving process which consumed a great deal of time and made the interactive theorem proving even more difficult.

Saving Procedure

When a model is changed and then saved, Rodin automatically runs the proof obligations generator to generate the proof obligations and prove them. However, there are times that there are numerous errors in the model that building the model is not the right choice. It would be more practical if whenever there are errors in the model, asking the user whether or not, he needs the model to be built. This could save a great deal of time and avoid peculiar error messages.

Hypotheses Relevance

A major difficulty with Rodin during the interactive theorem proving process was the huge number of hypotheses that were gathered automatically for some of the main proof obligations. It was extremely difficult to deal with this number of hypotheses that were listed for some of the proof obligations. Most of these hypotheses were not relevant to the specific PO at hand, but they were listed as the relevant ones. The average number of hypotheses for this model was about 50 which is relatively a large number for a model that is not very big. It was a difficult and also time consuming task to check all these hypotheses and determine which one is the most relevant to the proof obligation at hand.

In fact in chapter 5 some of the found patterns are mainly related to hypotheses and how a proof obligation can be easily discharged if the right hypotheses are present.

Error Messages

Rodin displays a number of different error messages for various different types of problems. However, these error messages are not very descriptive to understand their meaning as they are mostly acronyms. For example the error message for a missing bracket is "Syntax Error: RPAR Expected". Such non-self-explanatory error messages can be confusing and also time consuming to deal with for a new user. In comparison, in Java when there is a bracket missing or instead of a bracket, a curly bracket is found, the compiler displays the following error messages, respectively, "missing ) at line-number" or "( expected instead of { at line-number" which are much more intuitive.
6.10 Summary

In this chapter we explained some of the common challenges that a novice user is likely to face when approaching formal methods. Most of the challenges mentioned here were encountered during the course of this project and caused the progress of the project to be reduced. The Rodin tool-set was also evaluated based on its behaviour during the course of this project. Advantages and disadvantages of Rodin were discussed to highlight the practicality of this tool for assisting novice users. Our aim in this chapter was to provide novice users a guideline on their expectations when approaching formal methods.
Chapter 7

Conclusion

This project was mainly focused on formal methods from a software engineering prospective. A great deal of detail and mathematical work many not have been done in the short period of time available, but this project provides an example of how a novice user could approach formal methods and what are the important challenges that might discourage a user from such methods. This was not our initial aim of this project, however, because of the number of unforeseen challenges that had to be dealt with to complete the project, the direction was slightly changed and a different approach was chosen. In spite of changes that had to be made to the focus of the project, to focus on providing guidance and identifying reasoned modelling patterns, our main plan and strategy did not have to change fundamentally. We have provided some evidence for existence of patterns, we explained these patterns and it was shown that the difference that they can make to the modelling and reasoning processes is quite significant, specifically for novice users.

As it was presented, the existence of patterns in Event-B is quite common even for a small experiment such as this project. Their importance is quite significant in affecting the modelling process and in providing interactive help to the user. By discussing with some of the leading experts and participants at the VSTTE 2010 conference (such as Andrew Ireland), it was understood that the existence of modelling patterns is quite a common occurrence. The fact that a great deal of situations have been noticed by experts that a similar pattern had to be applied for a number of proof obligations or changes that had to be made to the segments of the model, which made the task quite tedious, was a common experience. This evidence provides a remarkable opportunity for improving formal methods, specifically theorem proving approaches. As mentioned before the work that is currently being undertaken by the AI4FM project is focused on finding patterns and increase the automation in theorem proving. However, there are a great deal of other kinds of patterns and guidance that could be offered to users such as reasoned modelling patterns, which in this project were remarkably useful. Also the idea of design patterns in software engineering could potentially be developed for the design phase of formal methods.

Based on the results of this project, offering users guidance from experts in formal methods, can make an extraordinary difference in the way new users deal with formal methods. Because one of the most important factors while using formal methods has been shown to be previous experience [20]. Not just previous experience in theorem proving, but also previous experience of the strategy that experts use to make decisions and represent requirements in mathematical notations. Such experience although might be based on opinions and personal preferences of the experts, can be of extreme value.
to a novice user when using formal methods.

A profound comment that was made by J Moore at VSTTE 10 about the challenges with formal methods and the current state of the field was that "we are mechanising thought" and this is not an easy task. It is important to note that according to Tom Ball’s keynote speech, creating correct software (by experts) is possible, if cost and time is not an issue. However, in order to make formal methods applicable to a wider range of software projects, it is required to solve such issues and to improve formal methods to a level that a single novice user can use such methods to create reliable software. This is important because an experience of the difference that formal methods can make, can be experienced by users of all kinds of backgrounds.

Note that what is meant by improving formal methods is not just creating better theorem provers and tools. Apart from such crucial improvements, it is important to create a discipline in using formal methods, something similar to what is promoted in software engineering to help the users in designing. A discipline that helps the user from the start to the end of the software production process.

7.1 Future Work

There are a great deal of opportunities for future work in this field. The list of works that can be done in the future are:

- Introduction of design patterns and a discipline in formal methods,
- Incorporation of patterns in theorem provers.
- Implementation of guidance in modelling.
- Creation of a platform that is a combination of the main ideas and tools in formal methods to help users in all phases of software production.

However, the future works mentioned above are quite challenging and need a very long research phase.

---

1At VSTTE 2010, Principal researcher at Microsoft Research http://research.microsoft.com/en-us/people/tball/
Bibliography


Appendix A

Tokeneer Context

The following context is the Tokeneer context, designed in this project.

**CONTEXT**  CTOKENEER

**SETS**
- Presence
- Clearances
- Privilege
- FingerPrint
- User
- Issuer
- KeyPart
- CertID
- TokenID
- Enrol
- Class
- tokenStatus
- EnclaveStatus
- Status
- TokenTry
- Fingerprint
- FingerprintTry
- Door
- Latch
- Alarm
- AdminOp
- AdminPrivilege

**CONSTANTS**
- time
- subject
- present
- absent
- privateKey
- publicKey
- certs
- idCert
- privCert
- iandACert
authCert
attCert
baseCertID
idCertInit
door latch alarm
open
closed
unlocked
locked
silent
alarming
class
unmarked
unclassified
restricted
confidential
secret
topsecret
tokentry fingerprint
noT
badT
noFP
badFP
goodF
Status
quiescent
gotUserToken
waitingFinger
gotFinger
waitingUpdateToken
waitingEntry
waitingRemoveTokenSuccess
waitingRemoveTokenFail
EnclaveStatus
notEnrolled
waitingEnrol
waitingEndEnrol
enclaveQuiescent
gotAdminToken
waitingRemoveAdminTokenFail
waitingStartAdminOp
waitingFinishAdminOp
shutdown
AdminOp
archiveLog
updateConfigData
overrideLock
shutdownOp
guard
auditManager
securityOfficer
initialisation
initvalidid
initFPTry
initFingerprint
initTokenTry
initPresence
AXIOMS

ax1: TokenID ≠ ∅
ax2: CertID ≠ ∅
ax3: User ≠ ∅
ax4: CertID ≠ ∅
ax5: Issuer ≠ ∅
ax6: Privilege ≠ ∅
ax7: Presence ≠ ∅
ax8: time = N
ax9: certs ⊆ CertID
ax9b: certs ≠ ∅
ax10: partition(Presence, {present}, {absent}, {initPresence})
ax11: partition(KeyPart, privateKey, publicKey)
ax12: partition(certs, idCert, attCert)
ax12a: idCert ≠ ∅
ax12b: attCert ≠ ∅
ax13: partition(attCert, privCert, iandACert, authCert)
ax14: subject ∈ CertID → User
ax15: baseCertID ∈ certs → CertID
ax16: partition(Door, {open}, {closed})
ax17: partition(Latch, {locked}, {unlocked})
ax18: partition(Alarm, {silent}, {alarming})
ax19: Enrol ≠ ∅
ax20: partition(Class, {unmarked}, {unclassified}, {restricted}, {confidential}, {secret}, {topsecret})
ax21: partition(TokenTry, {badT}, {noT}, {initTokenTry})
ax22: partition(Fingerprint, {initFingerprint})
ax23: partition(Status, {quiescent}, {gotUserToken}, {waitingFinger}, {gotFinger}, {waitingUpdateToken}, {waitingEntry}, {waitingRemoveTokenSuccess}, {waitingRemoveTokenFail})
ax25: partition(AdminOp, {archiveLog}, {updateConfigData}, {overrideLock}, {shutdownOp})
ax26: partition(AdminPrivilege, {guard}, {securityOfficer}, {auditManager})
ax27: partition(FingerprintTry, {badFP}, {noFP}, {initFPTry}, {goodF})
ax28: idCertInit ∈ idCert ∨
initax1: initvalidid ∈ (P(TokenID) \ ∅)
initthm1: initvalidid ⊆ TokenID ∧ initvalidid ≠ ∅
initax2: initClearances ∈ Clearances
initax3: initCertID ∈ (P(CertID) \ ∅)
\textit{initthm2}: \textit{initCertID} \subseteq \text{CertID} \land \textit{initCertID} \neq \emptyset \\
\textit{initax4}: \textit{initValPeriod} \in (\mathbb{P}(\text{CertID}) \setminus \emptyset) \\
\textit{initthm3}: \textit{initValPeriod} \subseteq \text{CertID} \land \textit{initValPeriod} \neq \emptyset \\
\textit{initax5}: \textit{initAuthCert} \in (\mathbb{P}(\text{authCert}) \setminus \emptyset) \\
\textit{initthm4}: \textit{initAuthCert} \subseteq \text{authCert} \land \textit{initAuthCert} \neq \emptyset \\
\textit{initax6}: \textit{initPrivilege} \in (\mathbb{P}(\text{Privilege}) \setminus \emptyset) \\
\textit{initthm5}: \textit{initPrivilege} \subseteq \text{Privilege} \land \textit{initPrivilege} \neq \emptyset \\
\textbf{END}
Appendix B

Tokeneer Machine

The following model is the Tokeneer machine which was modelled in this project.

MACHINE MTokeneer
SEES CTokeneer
VARIABLES
  ID
  valPeriod
  isValBy
  role
  clearance
  issuerKey  Certs And tokens
  tokenID
  tokIdC
  tokAttC
  tokPrivC
  tokAuthC
  tokIAC
  validToken
  attCertTokID
  currentTok
  idStationCert
  issuerCerts
  now
  validTokenID
  tokenWithValidAuth
  theAuthCert
  tokWithValidAuthID  config
  latchUnlockDuration
  alarmSilentDuration
  tokenRemovalDuration
  enclaveClearance
  authPeriod
  entryPeriod
  minPreservedLogSize
  maxSupportedLogSize
  alarmThresholdSize  latch door alarm
  currentTime
currentDoor
currentLatch
doorAlarm
latchTimeout
alarmTimeout
user Token token try
currentUserToken
userTokenPresence
goodT
goodFP
currentFinger
fingerPresence
status business
status
enclaveStatus
tokenRemovalTimeout
rolePresent
availableOps
currentAdminOp
currentAdminToken

INVARIANTS

inv1: ID ∈ certs → CertID
inv2: valPeriod ∈ CertID → P(time)
inv3: isValBy ∈ ID → KeyPart
inv4: tokenID ⊆ TokenID
inv4b: tokenID ≠ ∅
inv5: role ∈ CertID → Privilege
inv6: clearance ∈ CertID → Clearances
inv7: tokIdC ∈ tokenID → idCert
inv8: tokPrivC ∈ tokenID → privCert
inv9: tokIAC ∈ tokenID → iandACert
inv10: tokAuthC ∈ tokenID → authCert
inv11: tokAttC ∈ tokenID → attCert
inv12: attCertTokID ∈ attCert → tokenID
inv13: validToken ⊆ tokenID ∧ (∀x · x ∈ tokenID ⇒
               (x ∈ validToken ⇔ ID(tokIdC(x)) = ID(tokPrivC(x)) ∧ ID(tokIdC(x)) =
                ID(tokIAC(x)) ∧ attCertTokID(tokAttC(x) = x))
inv14: now ∈ time
inv15: currentTok ⊆ validToken ∧ (∀x · x ∈ validToken ⇒ now ∈ valPeriod(tokIdC(x)))

fix
inv16: idStationCert ∈ Enrol → idCert
inv17: issuerCerts ∈ Enrol → P(idCert)
inv18: enclaveClearance ∈ Clearances
inv19: latchUnlockDuration ∈ N
inv19b: latchUnlockDuration < 1000
inv20: alarmSilentDuration ∈ time
inv21: tokenRemovalDuration ∈ time
inv22: authPeriod ∈ Privilege → (time → P(time))
inv23: entryPeriod ∈ Privilege → (Class → P(time))
inv24: minPreservedLogSize ∈ N
inv25: alarmThresholdSize ∈ N
inv26: maxSupportedLogSize ∈ N
inv27: alarmThresholdSize < minPreservedLogSize
inv28: minPreservedLogSize ≤ maxSupportedLogSize
inv29: issuerKey ∈ Issuer → KeyPart
inv30: currentTime ∈ N
inv30b: currentTime < 1000
inv31: currentDoor ∈ Door
inv32: currentLatch ∈ Latch
inv33: doorAlarm ∈ Alarm
inv34: latchTimeout ∈ N
inv34b: latchTimeout ≤ 2000
inv35: alarmTimeout ∈ time
inv36: currentLatch = locked ⇔ currentTime ≥ latchTimeout
inv37: doorAlarm = alarming ⇔ (currentDoor = open ∧ currentLatch = locked ∧ currentTime ≥ alarmTimeout)
inv38: goodT ∈ tokenID → TokenTry
inv39: currentUserToken ∈ TokenTry
inv40: userTokenPresence ∈ Presence
inv41: goodFP ∈ Fingerprint → FingerprintTry
inv42: currentFinger ∈ Fingerprint
inv43: fingerPresence ∈ Presence
inv45: status ∈ Status
inv46: enclaveStatus ∈ EnclaveStatus
inv47: tokenRemovalTimeout ∈ time
inv48: validTokenID ∈ validToken → TokenID
inv49: theAuthCert ∈ authCert → tokenID
inv50: tokenWithValidAuth ⊆ tokenID
inv51: tokenWithValidAuthID ∈ tokenWithValidAuth → TokenID
inv52: status ∈ {gotFinger, waitingFinger, waitingUpdateToken, waitingEntry}
        ⇒ ((∃vt · goodT(validTokenID(vt)) = currentUserToken) ∨ (∃twva ·
goodT(tokenWithValidAuthID(twva)) = currentUserToken))
inv53: availableOps ∈ P(AdminOp)
inv54: currentAdminOp ∈ tokenID → AdminOp
inv55: rolePresent ∈ tokenID → AdminPrivilege
inv56: rolePresent = ∅ ⇒ availableOps = ∅
inv57: enclaveStatus ∈ {waitingStartAdminOp, waitingFinishAdminOp}
        ⇔ currentAdminOp = ∅
inv58: currentAdminToken ∈ TokenTry
inv59: enclaveStatus = gotAdminToken ⇒ rolePresent = ∅
inv60: enclaveStatus = ∅ ⇒ (∃x · x ∈
tokenWithValidAuth ∧ (goodT(x) = currentAdminToken))

EVENTS
Initialisation
begin
  init1: ID ∈ certs ⇒ initCertID
  fixed inv1
  init2: valPeriod ∈ initValPeriod ⇒ P(time)
  init3: isValBy := ∅
\text{init4} : \text{tokenId} := \text{initvalidid} \\
\quad \text{fixed inv3} \\
\text{init5} : \text{role} := \emptyset \\
\text{init6} : \text{clearance} := \emptyset \\
\text{init7} : \text{tokIdC} \in \text{initvalidid} \rightarrow \text{certs} \\
\quad \text{fixed inv50, 51} \\
\text{init8} : \text{tokPrivC} \in \text{initvalidid} \rightarrow \text{certs} \\
\text{init9} : \text{tokIAC} \in \text{initvalidid} \rightarrow \text{certs} \\
\text{init10} : \text{tokAuthC} := \emptyset \\
\text{init11} : \text{validToken} := \emptyset \\
\text{init12} : \text{attCertTokID} := \emptyset \\
\text{for getting tokenId of a Cert} \\
\text{init13} : \text{tokAttC} \in \text{initvalidid} \rightarrow \text{certs} \\
\text{init14} : \text{currentTok} := \emptyset \\
\text{init15} : \text{now} := 0 \\
\text{init16} : \text{issuerCerts} := \emptyset \\
\text{init17} : \text{idStationCert} := \emptyset \\
\text{init18} : \text{latchUnlockDuration} := 0 \\
\text{init19} : \text{alarmSilentDuration} := 0 \\
\text{init20} : \text{tokenRemovalDuration} := 0 \\
\text{init21} : \text{enclaveClearance} := \text{initClearances} \\
\text{init22} : \text{authPeriod} \in \text{initPrivilege} \rightarrow (\text{time} \rightarrow \mathbb{P}(\text{time})) \\
\text{init23} : \text{entryPeriod} \in \text{initPrivilege} \rightarrow (\text{Class} \rightarrow \mathbb{P}(\text{time})) \\
\text{init24} : \text{minPreservedLogSize} := 0 \\
\text{init25} : \text{alarmThresholdSize} := 0 \\
\text{init26} : \text{maxSupportedLogSize} := 0 \\
\text{init27} : \text{issuerKey} := \emptyset \\
\text{init28} : \text{currentTime} := 0 \\
\text{init29} : \text{currentDoor} := \text{closed} \\
\text{init30} : \text{currentLatch} := \text{locked} \\
\text{init31} : \text{doorAlarm} := \text{silent} \\
\text{init32} : \text{latchTimeout} := 0 \\
\text{init33} : \text{alarmTimeout} := 0 \\
\text{init34} : \text{currentUserToken} := \text{initTokenTry} \\
\text{init35} : \text{userTokenPresence} := \text{initPresence} \\
\text{init36} : \text{goodT} := \emptyset \\
\text{init37} : \text{currentFinger} := \text{initFingerprint} \\
\text{init38} : \text{fingerPresence} := \text{initPresence} \\
\text{init39} : \text{goodFP} := \emptyset \\
\text{init40} : \text{status} := \text{quiescent} \\
\text{init41} : \text{enclaveStatus} := \text{enclaveQuiescent} \\
\text{init42} : \text{validTokenID} := \emptyset \\
\text{init43} : \text{tokenWithValidAuth} := \emptyset \\
\text{init44} : \text{theAuthCert} \in \text{initAuthCert} \rightarrow \text{initvalidid} \\
\text{init45} : \text{tokWithValidAuthID} := \emptyset \\
\text{init46} : \text{tokenRemovalTimeout} := 0 \\
\text{init47} : \text{currentAdminOp} := \emptyset \\
\text{init48} : \text{rolePresent} := \emptyset \\
\text{init49} : \text{availableOps} := \emptyset \\
\text{init50} : \text{currentUserToken} := \text{initTokenTry} \\
\text{end} \\
\text{Event} \quad \text{readToken} \triangleq \\
\quad \text{any} \\
\quad \text{tok} \\
\quad \text{where} \\
\quad \text{grd1} : \text{tok} \in \text{tokenId} \\
\quad \text{grd2} : \text{status} \neq \text{gotUserToken}
EnclaveStatus ∈ {enclaveQuiescent, waitingRemoveAdminTokenFail}

then
  act1: currentTok := currentTok ∪ {tok}  
    my own extension
  act2: status := gotUserToken
  act3: userTokenPresence := present
end

Event tearToken =
  any tok
  where
    grd1: tok ∈ tokenID
    grd2: tok ∈ currentTok
    grd3: status ∈ {gotUserToken, waitingUpdateToken, waitingFinger, gotFinger, waitingEntry}
  then
    act1: currentTok := ⊘
    act2: status := quiescent
    act3: userTokenPresence := absent
end

Event noBioCheck =
  any tok
  where
    grd1: tok ∈ tokenID
    grd2: userTokenPresence = present
    grd3: status = gotUserToken
  then
    act1: status := waitingEntry
end

Event readFingerPrint =
  when
    grd1: status = waitingFinger
    grd2: fingerPresence = present
    grd3: userTokenPresence = present
  then
    act1: status := gotFinger
end

Event fingerTimeOut =
  when
    grd1: status = waitingFinger
    grd2: userTokenPresence = present
  then
    act1: status := waitingRemoveTokenFail
end

Event validateFinger =
  when
    grd1: goodF = goodFP(currentFinger)
    grd2: userTokenPresence = present
    grd3: status = gotFinger
  then
    act1: status := waitingUpdateToken
end

Event writingUserToken =
  any
tok
where
  grd1: tok ∈ tokenID
  grd2: userTokenPresence = present
  grd3: status = waitingUpdateToken
then
  act1: status := waitingEntry
end

Event allowEntry ≡
any
  tok
where
  grd1: tok ∈ tokenID
  grd2: userTokenPresence = present
  grd3: status = waitingEntry
  grd4: ∃vt· goodT(validTokenID(vt)) = currentUserToken
then
  act1: status := waitingRemoveTokenSuccess
  act2: tokenRemovalTimeout := currentTime + tokenRemovalDuration
end

Event unlockDoor ≡
when
  grd1: status = waitingRemoveTokenSuccess
  grd2: userTokenPresence = absent
then
  act1: latchTimeout := currentTime + latchUnlockDuration
  act2: alarmTimeout := currentTime+latchUnlockDuration+alarmSilentDuration
  act3: currentTime := currentTime
  act4: status := quiescent
end

Event lockDoor ≡
begin
  act1: latchTimeout := currentTime
  act2: alarmTimeout := currentTime
  act3: currentLatch := locked
end

END