Using Behavioural Specifications to Support Model-Checking

A thesis

submitted in partial fulfilment

of the requirements for the degree

of

Master of Science (Research) in Computer Science

at

The University of Waikato

by

Bowen Liu

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2019

Abstract

Safety-critical interactive systems provide many benefits for human daily life, but erroneous safety-critical interactive systems can lead to serious consequences to the users. Thus building these systems requires that we ensure a high level of correctness. Formal models can be used to ensure that safety-critical systems are developed correctly. However, when models of the systems are being built, there is no guarantee that the modelled system fully satisfies the user requirements. If the systems are not what business stakeholders truly desire, new errors can be made.

Behaviour-Driven-Development is often applied to ensure the requirements of the system are properly understood and maintained by using Behavioural Specifications. These specifications use natural language, and so, are well suited for expressing user requirements. They can be used as the basis for test generation but do not provide the guarantees of correctness that formal methods can provide. In this work, we develop an approach that takes advantage of the expressivity of behaviour specifications and combines it with the use of formal methods. We use Behavioural Specifications to create First-Order-Logic predicates of the requirements. These predicates can be used with formal methods, either to support the creation of the formal models or to ensure the user requirements and specifications are consistent.

Acknowledgement

I would like to express my sincere appreciation to my supervisor Doctor Judy Bowen for her continued support, patience, and encouragement. Your guide and kind words throughout this year of researching and writing helped me to build up the knowledge from the ground up. I wouldn't have finished this research without your continuous comments and teaching. Thank you for enduring my slow start and numerous questions. In addition to my supervisor, I would like to thank my colleagues at the University of Waikato. I have learned a lot from you and I really appreciate all your encouragement. I am also grateful to University of Waikato, for helping me over the 6 years and granting me all the scholarships. Lastly, I would like to thank my family. I would like to say thank you to my mom Lv Hong, I wouldn't have got there without her 23-year-long enormous love and support, days and nights, you helped me through a lot. I would thank my partner, Xie Huiling, for coming to New Zealand with me. I would also say thank you to all my other family members and friends who gave me support over the years. I wouldn't never have finished this research without you.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Interactive computer systems are computer systems that support real-time interactions between humans and computers. Interactive computer systems require human interactions and provide information for users in return. These interactions are facilitated by user interfaces and/or widgets, then the information can be the output of these systems, such as an alert box from the user interface or some images displayed in a display widget. Interactive computer systems are often used to improve efficiency and effectiveness in order to reduce common costs of daily tasks. When we build interactive systems, we must consider not only the functionality and correctness of the software, but also the usability, for example, how robust the interactions are and how easy it is to use the system.

Safety-critical interactive systems are a special type of interactive system, where errors can have serious consequences. For example, bank systems help people to manage their finances, so the systems must be correct and utilisable, otherwise the fiances of the user will be damaged. Another example is that of medical software and devices that help with supporting human health, erroneous systems can do harm to the user or even kill them. While safety-

critical interactive systems play an important role in improving the quality of life, developing and delivering these systems require very high correctness and robustness. For example, a faulty infusion pump could kill the patient when it is used for treatment if it fails to deliver medications properly [23]. In addition to correctness, we need to consider usability, interactive systems need to be easy to learn and use, over-complicated design can distract the user and lead to human errors which might also result in the death of the patient. For example, an infusion pump that requires 27 steps to correctly operate has a very high chance to cause accident [41].

When we develop safety-critical systems, it is possible that the business stakeholder who gives requirements knows very little about software development, therefore it is important to have a clear set of unambiguous requirements. On the other side, the requirements could be very domain-specific, for example, the system developer might not understand how an infusion pump works or the medical domain where it is used. Thus the possibility of having miscommunication between business stakeholders and professional software developers is too high to ignore. In addition to the communication problem, consistency needs to be maintained throughout the whole implementation process, the developer needs to carefully stick to the original requirements as the system grows bigger and more complicated.

Test-Driven Development (TDD) [16] is a popular software development approach where tests are written before the code. TDD's core development process is simple: Write a test that should fail; Code is modified to make the test pass; Add new tests and refactor the code. However, this simple but effective approach suffers from ambiguity - the acceptance tests rely on the testers to infer and write correct tests from the given requirements. If the testers misunderstood the requirements, the outcome could be not as expected. Delivering software that fully satisfies the requirements can be very difficult and expensive.

Given this communication gap, Dan North [50] proposed Behaviour Driven Development (BDD), an agile software development process that emerged from Test-Driven Development. Similar to TDD, the BDD approach starts with writing semi-formal Behavioural Specifications. A Behavioural Specification [62] describes the business and user requirements for software systems being developed in a structured natural language based around scenarios of use. Behavioural Specifications (following some rules or writing conventions) can be scenarios (user stories), use cases or plain descriptive language that describes how the application is expected to run. Such a specification serves as the basis of tests (even though requirements are given with behaviour, there still needs to be acceptance tests as a benchmark). Using behavioural specification as a guide for development can increase the clarity of the requirements.

Formal methods [22] are techniques that use formal mathematical reasoning or logic for the specification, development and verification of computer software and hardware systems. Such methods include Model-Driven-Development (MDD) and Model-Driven-Engineering (MDE). Model-Driven-Development is a formal software development approach that uses a range of domain models to represent the business requirements instead of using purely textual descriptions. These abstract models allow the software developers to have a better understanding of the definition of the system that is being developed, and thus the developers will have a better chance to deliver the correct system that the stakeholders actually need. With formal modelling, the correctness and usability of the safety-critical systems can be improved greatly.

Similar to Test-Driven-Development, in the MDE domain, the models fully rely on the modellers to understand what the requirements actually are and correctly translate the requirements to useful models. There could be a difference between the knowledge that stakeholders possess and the model checking experts have, and thus the communication gap between model checking experts and stakeholders can be even bigger than the gap in a typical TDD

development environment. Since modelling languages are hard to learn and we don't expect the stakeholders to get a full understanding of what the translated models should look like, the typical modeling approach fully relies on the model checking experts to correctly infer, transform, build and maintain the models from the requirements.
In a typical software development environment, there are two major problems: The informal business requirements may be ambiguous and therefore hard for the developers to correctly understand (unintentionally, but it does affect how the developers deliver the software). Even if the developers have correctly understood the requirements, correctly constructing models while maintaining the requirements throughout the whole implementation progress is difficult. Our question is, what can help to improve such a problematic process?

Cucumber [54] is a software testing tool that supports tests written in a plain logical language called Gherkin. Gherkin follows the Behaviour-Driven Development (BDD) [50] style and it is typically used as a connection between business stakeholders and software developers. One important feature it has (which is also what we care about the most) is unambiguous executable specification - Cucumber Scenarios. The scenarios indicate the expected behaviours of the software being developed. Since Cucumber Scenarios are written using natural structured language, the scenarios (such as "A vending machine takes 2 dollars and provides a can of L&P") allow both stakeholders and developers to understand what is required.

Cucumber Scenarios are usually used as base information for writing or generating tests. These tests can then be executed by the Cucumber tool to support development. However, we aim to use the Cucumber Scenarios in a different way - to support model-checking for models written with a formal notation.
Z [30] is a formal mathematical specification language that can be used for precisely specifying and modeling functional behaviours or structures. The Z

notation is not specifically designed for model-checking, it is neither a programming language nor a natural language. Thus "writing models" with Z notation here is actually writing some formal specification of how software should behave. Then these behaviours (Z operations) can be used with theorem provers to formally prove properties of the described system, or transformed into models in some other language (the B method in our case), or refined into code. The mathematical aspect of Z is based upon First-Order Logic. The constraints and declarations can be written using First-Order Logic predicates.

As we discussed, for safety-critical interactive systems, correctness is the most important thing we need to consider, as errors can lead to serious consequences for the users. When business stakeholders give out requirements, these requirements can be prototypes or behaviour specifications. These requirements will then be used by the developers to create formal specifications and models. The problem is that there is a gap between formal models and requirements. Formal languages such as Z require expertise, they are not intended to be understood by business stakeholders, we can't ensure that the requirements are understood correctly by the developers and the modelled system meets the defined pre-requisites. There is no guarantee that the built system is what the stakeholders actually desire, which could cause new errors. Thus we need to think of a way to connect the stakeholders and model checkers, so that the correctness of the system is ensured.

To fill this gap, we propose the following method: Start with the Cucumber Scenarios written by the business stakeholders, we then transform the scenarios into First-Order Logic predicates. These First-Order Logic predicates can then be used as a subset of the predicates for the Z specification. The Z notation schemas will be used to form the models of the desired system. The more formal Z models can be verified/model checked later.

Behaviour specifications are given by the business stakeholders, which ensures the requirements are correct. By transforming the behaviour specifications,

the predicates computed are intended to be in accordance with the pre-defined requirements. Therefore, the formal Z specifications built that contain these predicates as constraints are also ensured to meet the requirements.

1.1 Contributions

This thesis proposes a way to transform requirements written as Behaviour Specifications into a more formal form. Using the more formal requirements to guide the modeling process within a formal software development process is safer, more robust and less ambiguous. Using behaviour specifications also means usability is improved. This approach can be used as a bridge to connect business stakeholders and software developers, allowing them to work in conjunction to deliver safety-critical interactive systems that are better in correctness, robustness and usability.

1.2 Thesis structure

In chapter two, we will discuss the background material and relevant literature that are related to our work, followed by a discussion of why they are related. We will particularly focus on explaining why using Formal Methods and Behaviour-Driven Development in conjunction is helpful to us.
In chapter three, we will discuss the methodology of the research. This will explain how we were planned to conduct this research, including how we selected the formal language and model checker for the approach.
In chapter four, we will introduce and explain the bank scenario example developed for this research: the example is given from the view of business stakeholders and model developers, including Cucumber Scenarios and a Z notation.

In chapter five, we will demonstrate how we implemented the approach. Starting with discussing the challenges we faced and initial attempts. This is followed by discussion on the 3 versions of the converter we built and a case study on applying the converter to the banking scenario. Finally, we test the converter with other scenarios and evaluate the results.

In chapter six, we summarise the report and discuss possible future works that can be done to further improve the approach.

Chapter 2

Background and Related work

2.1 Introduction

In this chapter we will introduce the software development methods and technologies related to this thesis and the relevant literature in the area of those technologies. The major goal of this thesis is to enhance model-driven development and improve interactive system development and so the focus here is an approach for developing interactive systems.

2.2 Formal Methods

As we introduced, **Formal Methods** [22] are techniques that use formal mathematical reasoning or logic to specify the desired behaviour of a system, the specification of the system is then verified. Formal methods can be applied at the earliest stages of development, they help with specifying the behaviours. The requirements are maintained throughout the development process, by us-

ing the formal specifications of the system as a guide for development. Formal methods can also be used for verification, the specification or formal model of the system is verified using different tools such as theorem provers. By verifying the formal specification, the correctness and robustness of the system being developed can be improved. The formal specification can then be used to refine into an actual implementation of the desired system.

For safety-critical interactive systems, errors can lead to very serious consequences, thus we care the most about correctness. Formal methods are useful in helping with ensuring correctness when we make use of the formal specification and verification techniques.

2.3 Model-Driven Development (MDD)

Model-Driven Development is a software development technique that focuses on building a range of abstract domain models to represent the desired system, before writing any code. The models specify how the desired system should work, thus they can be used as the basis for the software development, such as generating code in a programming language.

The domain models provide a better way for the software developers to understand the business requirements. More precise implementation of the requirements means the correctness of the system being built is improved, when we apply MDD to safety-critical interactive system development, this is particularly useful, as correctness is the most important aspect of safety-critical interactive systems.

2.4 Model-Based Testing (MBT)

Model-Based Testing (MBT) [61] is an approach of automated testing of applications using models as the basis for the tests. Abstract models are developed to describe the expected behaviour of the system, when a software is being tested, the run time behaviour of this software is checked against the output of the abstract model. A typical MBT process goes as followed: The developer implements the software and a model, which is a partial description of the software. Then another program sends the same input (input can be a representation of the desired behaviour of a system under test (SUT), or representation of testing strategies and a test environment) to the SUT and the model. The results will be compared and made sure they are the same. If they are not, the test is failed and the software needs to be improved. With this strategy, developers can keep creating new models and testing against them to improve the SUT.

Bowen and Reeves [19] have described a way of formally describing the meaning of informal design artefacts (such as personas, storyboards, scenarios or prototypes), called the presentation model. The presentation model can describe different windows of a UI and all the possible behaviours of the UI. The presentation model uses a conservative extension of set theory as its semantics, "that is, everything which is provable about Presentation Models from the semantics is already provable in set theory using the definitions given in the semantic equations"[19]. The presentation model can be used for:

1. Refinement, the developer should ensure the design of the UI is considered/included in the formal refining process when the system is developed.

2. Design Equivalence, different versions of the UI designs should be provable to the presentation model.

3. Design Consistency, the developer can use the presentation model to ensure the same functions have the same names.

To extend the information that the presentation model can hold, Finite State Machines (FSM) [49] are combined with the presentation model to present the dynamic behaviours of the UI, called Presentation and Interaction Model (PIM).

Later, Bowen and Reeves [21] proposed an approach that uses abstract tests (automatically derived from formal models of UIs and system specifications [20]) as the basis for Test-First development (TFD, where models of the requirements are used to generate tests prior to implementation). The approach is similar to the regular TDD approach: With requirements given, a set of models that consists of functional specification, user requirements, task analysis and prototypes (UI models) are developed. This set of models should satisfy all of the functional and user requirements. A set of abstract tests is then automatically derived from the UI models using the PIMed tool [39]. The two sets of models and abstract tests are the basis for the test-first process (corresponding to the tests in TDD process). Note that there are additional constraints given by the prototypes and UI models, different from the typical TDD process. The rest of the development steps is the same as in a TDD process - Write a test -> Check that test is failed -> Write enough code to pass the test -> Refactor. However, the abstract tests are used to determine the next test, the design and UI models are required to be satisfied throughout the whole process.

Hellmann et al. proposed an alternative approach for developing graphical

user interfaces (GUI) from low-fidelity prototypes [33, 34]. Details of user stories are collected and used to produce a low-fidelity prototype of the expected system. This prototype is then improved by repeatedly conducting usability evaluations and modifying the prototype to match interface requirements. Once the prototype is improved to be sufficiently stable, information regarding the expected behaviour is added to further improve the prototype. A capture-replay tool (CRT, a tool for Graphical User Interface (GUI) that records the interaction between user and application when the user runs this application, the tool can then replay all the interactions automatically without a human user. Hellmann et al. used LEET [32] in their case) is then used to produce complex acceptance tests. These acceptance tests and the stable prototype are used in conjunction as the basis of the TDD process to develop the targeted GUI.

Applying MBT allows the developers to find bugs regarding design and specification when they build abstract models, this helps to improve the correctness of the system under development and save time/effort before any actual code is written. In safety-critical interactive system development, using MBT with formal method languages (Z notation in our case) can help in reducing costs and improving correctness.

2.5 Test-Driven Development (TDD)

Test-Driven Development [16] is a software development process where the developer repeats a very short development cycle: write initially failing tests for the software and then writing minimum code necessary to make the tests pass. TDD was "rediscovered" by Kent Beck in 2003 and he defined the process as: "1. Quickly add a test 2. Run all tests and see the new one fail 3. Make a

little change 4. Run all tests and see them all succeed 5. Refactor to remove duplication"[16].
Automated testing is an approach that uses special software (different from the software being tested) to continuously conduct tests via automation. Typically, Test-Driven Development and Behaviour Driven Development (variant of TDD that we will discuss later) use automated testing as their testing method.

Using Test-Driven Development will produce a lot of tests that are ensured to pass, which increases the correctness of the developed system. This is useful for safety-critical interactive system development. But TDD also suffers from the problem that the tests fully rely on the testers to correctly infer tests from the given requirements, this is similar to the problem we face when safety-critical interactive systems are built: The developed system might not fully satisfy the given requirements. In order to solve this problem, Dan North [50] proposed Behaviour Driven Development.

2.6 Behaviour Driven Development (BDD)

In 2003, Agiledox, which was believed to be the ancestor of BDD, was created by Stevenson [57]. Agiledox is a tool that automatically generates simple documentation from the method names in JUnit test cases (now Agiledox is named Textdox).
Later in 2003, North in collaboration with Matts [50], introduced Behaviour Driven Development which is an agile software development process that emerged from Test Driven Development. BDD combines the practices, techniques and principles of Test Driven Development (TDD) [16], Acceptance Test Driven Development (ATDD) [1] and Domain-Driven Design (DDD) [11].

BDD "is about implementing an application by describing its behaviour from

the perspective of its stakeholders" [51]. When developers apply BDD, they are still writing tests, but these "tests" are explained as behaviours of applications, which is more user-focused. Behaviour Specifications are written in descriptive plain English, so BDD can utilise natural language that non technical stakeholders can understand (what behaviour they expect that the application can do). Also, BDD is driven by business value, things are "outside in" which means implementing only those behaviours, which contribute most directly to these business outcomes, in order to minimise waste.

Carter and Gardner introduced a development approach called BHive [24], which combines BDD and B-method [13] (a tool-supported formal method). Carter addressed 3 classes of failure in the development of complex software systems: Failure to deliver, Catastrophic failure and Failure to maintain. In the discussion of Agile and Formal methods, Carter deemed that each of these two development methodologies were created to solve one of the 3 classes of failures. Thus BHive, an approach that avoids delivery risks while keeping high standard of correctness, is introduced.
The goal of BHive is to integrate traditional BDD development process with formal methods to assure/improve correctness of the application. BHive generates a B-machine based on Cucumber (a BDD tool that will be further discussed later) features by translating the Given-When-Then scenarios. The B-machine captures the expected behaviour of the system under development for the developer team, thus this model can be verified and used to build new tests. Although this approach was prototyped using Python's "Behave" [17] module, BHive is generalisable to any BDD tooling.

Lübke and Lessen [47] applied BDD to their project for reducing land registries process execution time where Business Process Model and Notation (BPMN) [3] was used as the test case model instead of normal text-based BDD business-readable domain-specific language (DSL) [31] such as "Given the user is logged

in". The test case models are fed into a generator along with content mappings and assertion mappings. The generator generates executable test suite automatically based on these BDD artefacts.

Laz˘ar et al. [43] attempted to combine Model-Driven Development (MDD) and BDD by defining "a UML profile that allows developers to build foundational UML (fUML) [7] models using a BDD approach" and "a BDD library containing activities that can help users to build executable scenarios." They also presented a tool called bUML which supports all BDD activities and allow users to create fUML models based on the proposed UML profile and library for BDD.

Hellmann introduced a tool called LEET [32] based on Microsoft's User Interface Automation Framework. LEET can be used to record manual exploratory test (a "style of software testing" [40]) sessions and apply automated rule-based (a automated testing method that reports behaviours which don't follow the rule) verification to them. First, the interactions performed during an exploratory test session were recorded as a re-playable script. A set of rules are defined to restrict the behaviours of the application. The set of rules and script are used to produce an automated regression test that can be used as the basis for standard TDD procedure. With this approach, the typical difficulty of creating models that can be automatically tested for manually created tests is avoided.

BDD aims at strengthening the understanding of user requirements by describing the expected behaviours of the system. BDD can be used to help the developers to better understand the requirements of safety-critical interactive systems, this is particularly important because using BDD means tests don't fully rely on software developers to correctly infer from the requirements. Formal Methods can use these BDD behaviour specifications to help with con-

structing their models thus the correctness of the safety-critical systems are further improved.

2.7 Cucumber/Gherkin

Cucumber [54] is a software testing tool that can run automated acceptance tests written in BDD style. Cucumber uses its own language Gherkin which describes expected software behaviours in plain language that consumers can read and write. The business stakeholders can thus understand and deliver more business-facing documentation. The connection between computer programmer and stakeholders is enhanced by this style.

Li et al. [45] introduced a Model-Based Testing (MBT) tool called Skyfire. Skyfire reads EMF (Eclipse Modeling Framework) [56]-based UML behavioural models and identifies model elements. Then these UML models are converted to general graphs. With a specified graph coverage criterion, Skyfire generates test paths that consist of graph nodes and edges. Abstract tests are then generated to map the test path to the UML machine diagram. Finally, abstract tests are converted to Cucumber test scenarios used at the User Acceptance Testing (UAT) level. A set of abstract tests derived from a single behavioural model is included in a Cucumber feature file where each abstract test is converted to a Cucumber test.

Colombo et al. [27] introduced a restricted approach to combine related scenarios into models which can be consumed by Model-Based Testing tools. The research was restricted in web-application area, which means it can't be applied to other contexts without modifying.
This approach adopts three conventions when writing Cucumber Scenarios: Make states explicit, Identify start states and Identify actions, preconditions

and post-conditions. If these conventions are adhered to, the three keywords (Given, When, Then) can be used to specify different components of a QuickCheck model [25]. A model can be constructed using this technique by processing multiple test scenarios and combining the results.

Snook et al. [55] introduced an approach that aims at applying Behaviour-Driven Development (BDD) principles to formal systems modelling and validation. It starts by producing a safe model (fully proven to be consistent) based on manually written scenarios using Event-B/iUMLB (Event-B is a formal method for system-level modelling and analysis; iUMLB is a Graphical front-end, a collection of diagrammatic editors for Event-B). Then this model is verified against manually written Cucumber Scenarios which can be used as input for a scenario generator. The scenarios generated by this generator are used for acceptance testing of the verified model and illustrated by Cucumber for Event-B/iUML-B. This approach used suggested "Cucumber for Event-B" which allows people to execute the Gherkin scenario directly in Event-B and "Cucumber for iUML-B" which provides Gherkin syntax to validate iUML-B class diagrams.

Khanal and Bowen proposed an approach that combines interactive system models (Presentation Models [19, 39]) and Behavioural Specifications (Cucumber Scenarios [46]) to automatically generate test stubs [18]. Given that Cucumber Scenarios can be automatically converted into test stubs, Khanal created an automatic process that transforms Presentation Models created by PIMed tool [39] into Cucumber Scenarios by applying transformation conventions. Although the transformed language of the Scenarios is more abstract, the initial meaning of the models is proved to be preserved. Further more, Khanal extended the PIMed tool so it can automatically generate limited Presentation Models from Cucumber Scenarios, but this transformation requires restriction on Cucumber Syntax.

The advantage of using Cucumber in our research is that the requirements presented are easier to understand, this helps the developers to get more precise implementation. However, Cucumber uses natural language whose meaning heavily relies on the grammar that the scenario writer is using, this increases the difficulty of automatically transforming Cucumber Scenarios into formal models. Therefore, we need to come up with a way to solve this problem.

2.8 Z

Z [30] is a formal mathematical specification language that can be used to describe functional behaviour or structure. Z was proposed by Abrial at Oxford University in 1980s. Z notation is based on high-level mathematical notations (such as sets, relations, functions, schema). Z specification models can be model checked with ProZ [52] tool (extension of ProB [44]). One of Z's advantages is that Z notation uses an expressive and precise structure which is unambiguous, this is especially good for formally expressing the functional aspects of the system without causing confusion.

Cristiá and Monetti introduced a Model-Based Testing tool called Fastest [28] that automates test suite generation and test case derivation from Z notation models for unit testing, by implementing Test Template Framework (TTF) [58]. Users start with applying some custom testing tactics to each operation in the Z model, after that, Fastest automatically constructs each testing tree node by node based on each custom testing tactic, by manipulating the Z text. The testing trees can also be manually pruned.

Plagge and Leuschel developed **ProZ** [52] which is an extension of ProB [44] animator and model checker (a model checker for B notation developed by Jean-Raymond Abrial) to support Z specifications. ProZ uses the Fuzz Type

Checker [8] by Spivey for extracting the formal specification from a LaTeX file. When ProZ loads a Z specification, ProZ translates Z to B specifications. One key difference between Z and B is that Z explain the purpose of each schema in its title, while B divides the whole specification into multiple sections using keywords such as VARIABLE. Another difference is that in B invariant is a constraint that must hold true in every state, while in Z invariants are predicates of the state schema which is implicitly held true in the operations. Thus violation of the invariant in B will lead to an operation being disabled in Z. When ProZ is used, a latex file containing Z specifications is taken into fuzz typechecker and parsed into a syntax tree. ProZ compiler takes in the syntax tree and translates it into an internal representation of a B machine. After that, ProB can perform model check on the B machine.

Z specifications are precise and unambiguous, using Z specifications to present safety-critical systems is useful in improving the system's correctness, thus we considered transforming Cucumber Scenarios into Z specifications in this research, along with ProZ as the model checker for checking Z specifications.

2.9 Alloy

Inspired by the Z specification language [30] and Tarski's relational calculus [60], Jackson et al. created Alloy [29]. Alloy is a descriptive specification language and it comes with an analyser (also called Alloy). Alloy can be used for describing behaviours, constraints, structures in the form of a model and its analyser can then explore the model, such as finding counter-examples in the specification. The analyser can visualise the specification model as graphical representations, which is easier to understand by the users. Alloy has been deployed in a wide range of applications such as auctions [59], electronic

commerce [53] and security [12].

A key feature that Alloy has is that its analyser can only perform a finite scope check on the model being explored, the user needs to choose the number of elements in each primitive types to be searched. The finite scope check means that Alloy makes no guarantee about the property being checked holds for larger scopes [29]. However, designers of Alloy analyser justify [6, 38] that high proportion of bugs of in the models can be found by examining within some small scope, according to the so-called "small scope hypothesis" [14].

Alloy can be used to represent systems under development as models and visualise them, this is useful in helping developers to understand the requirements of the system. The relatively smaller finite scope checks can be used to find bugs in the models. However, the correctness can't be guaranteed. Using Alloy in safety-critical interactive system development can improve the usability of the system, we considered using Alloy and its analyser in our research.

2.10 Summary

In this chapter, we discussed Formal Methods development methodologies including Model-Driven-Development and Model-Based Testing and some formal languages including Z and Alloy, these two languages will be considered for our research - the behaviour specifications will be used to support either Z or Alloy specifications. We discussed how relevant applying these Formal Method techniques is to safety-critical interactive systems as in advantages and disadvantages. Furthermore, we discussed how Test-Driven Development and Behaviour-Driven Development are related to safety-critical interactive system development, in particular we discussed Cucumber, a tool that supports Behaviour-Driven Development, this tool can be used to specify the expected

behaviours of the interactive systems by scenarios. Cucumber will be used for specifying behaviours of the systems in our research.

Based on the work we considered, we found that formal models [19, 21, 33, 34] can be used to support the system development. It is possible to combine BDD principles with formal systems modelling [13, 47, 43, 32, 45, 55] to strengthen software development. It is also possible to combine formal methods and behaviour specifications to support automatic acceptance test generation [18, 28]. These findings mean it is possible and useful to combine Behaviour Specifications and Formal Methods, in order to support interactive system development, thus we considered how Behaviour Specifications can be automatically transformed into formal models, or some useful elements that can support the formal models construction.

The common issues of automating Cucumber Scenarios transformation are that the scenarios use natural language and the meaning of the scenarios rely on the grammar of the writer. There was work done in an attempt to automatically transform Behaviour Specifications into models [27], this work was restricted in the web-application area, but it gave us the idea that we can set up rules when writing Cucumber Scenarios. The rules increased the difficulty of writing Cucumber Scenarios, but they also make the transformation from scenarios to models easier.

Based on investigation of the related work we considered above, we found that there is no existing techniques that can automatically transform Cucumber Scenarios into formal Z specification or Alloy model, or any element that can be used to support these formal methods. Therefore, we propose our approach - automatic transformation from Cucumber Scenarios to first-order logic predicates, that can be used to support Z specification or Alloy model construction.

Chapter 3

Methodology

3.1 Introduction

In this chapter, I will start with introducing the formal languages and their model checkers, and explain the choice of Z as the final formal language and ProB (ProZ) as the final model checker for this research. Next, I will discuss the idea of transforming Cucumber Scenarios to First-Order-Logic predicates, including the plan of the research and structure of the expected process.

3.2 Selecting formal language

At the start of the research, I was looking for predicate-based formal languages that have model checkers to support them. Z was chosen as the final formal language with ProB (ProZ) as its model checker. As I have already introduced Z and Alloy in the Background and Related Work chapter, in this section I will compare these two languages and discuss the difference and similarity between

these two model checkers. Finally, I will explain why I chose Z and ProB.

Alloy analyser is a model checker developed specifically for the Alloy language. ProZ is an extension of ProB model checker built for Z specifications. In ProZ, Z specifications are translated into B methods, which includes a state-based automata representation, in order to be model checked. Note that we are using standard Z as described by ISO in 2002 [35].

Z has a major influence on Alloy, referring to the author of Alloy [36], Alloy can roughly be viewed as a subset of Z. Z and Alloy are not programming languages, they are both **declarative** languages. Z and Alloy can describe a behaviour, operation or function with syntactic constructs. These constructs can be the declaration (such as names of the operations) of the operations or some constraints specifying input and output of the operations being described (such as insert a coin, drop a can of soda). Alloy uses *signatures* for declarations [37] (names, properties), and *functions/predicates/facts* for functionalities/constraints. Different from Alloy, Z notation uses only *schema* to express the declarations and functions.

Here is the example for the same Deposit Money operation of a banking system written in Alloy language (Figure 3.1) and Z notation (Figure 3.2):

```
pred DepositMoney
    [B,B': BankAccount,
    Amount_in: Int]
    {
    B.AccountType != New
    Amount_in > 0
    Amount_in < AmountLimit
    B'.Credit = B.Credit + Amount
    B'.OperationToday' = B.OperationToday + 1
    B'.AccountType = B.AccountType
    }
```

Figure 3.1: DepositMoney operation in Alloy

DepositMoney

$\Delta BankAccount$

$Amount? : \mathbb{N}$

$\neg(AccountType = New)$

$Amount? > 0$

$Amount? < AmountLimit$

$Credit' = Credit + Amount?$

$OperationToday' = OperationToday + 1$

$AccountType' = AccountType$

Figure 3.2: DepositMoney operation in Z

In the above examples we can see that the Alloy DepositMoney (Figure 3.1) and Z DepositMoney (Figure 3.2) operations are nearly identical, the operation is done on the BankAccount with Amount/Amount_in as the operation parameter. The AccountType of the BankAccount can't be New, the Amount/Amount_in need to be smaller than AmountLimit and greater than 0. The new Credit of the BankAccount is the sum of old Credit and the Amount/Amount_in, the AccountType stays the same and the OperationToday of the BankAccount is increased by 1.
We can see one difference between these two specifications is that Alloy used integers for Amount_in while Z used natural numbers set, this is because Alloy doesn't support direct reference to natural numbers set, but this can be easily fixed by adding a constraint that Amount_in must be greater than 0. Another difference is that Z uses only the prime decoration ("′" symbol) after the variable for declaring the variable value after the state change, while Alloy uses B′.variable name for declaring the variable value. In this case Alloy has an arguably clearer presentation.

Alloy's composition mechanisms are flexible like the schema calculus that Z language schemas use, but "they are based on different idioms: the model extends by adding more fields; similar to inheritance in an object-oriented language, and reuse of formulas by explicit parameterisation, similar to functions in a functional programming language" [37].

Note that Alloy language is purely based on ASCII which means writing Alloy models doesn't require any special typesetting tools. In contrast, Z notation uses many non-ASCII symbols which can't be directly written using a keyboard. But Z notation is typically written with LaTeX[42], a document preparation system, which makes writing non-ASCII standard Z symbols easy and convenient.
Additionally, Alloy supports graphical object models which allows the users

to have a direct view of the models being built. In ProZ, this can be done by installing AT&T's Graphviz package [9].
Alloy does not provide direct support for many abstract data types, there are only strings, booleans and integers while Z notation supports many abstract mathematical data types.

The motivation of Alloy (which is also the main difference between Alloy and Z) is to allow fully automated analysing of the models built. Z was lacking this function which means abstract models built with Z notation can't be directly model checked. Fortunately, this problem is solved by translating Z specifications to B machines which can then be model checked in ProB [44]. In fact, Malik et al. [48] has developed an approach to translate Z specifications to Alloy models. But this is irrelevant so we won't discuss it further in this report.
As we mentioned before, an important feature of Alloy is that it only allows finite scope checks on relatively small models, this feature increases the speed of the model checking process, but the correctness can't be guaranteed. Model checking Z specifications using ProB allows exhaustive research on the models, which ensures correctness.

As we discussed, the most important aspect of safety-critical systems is correctness. Due to the fact that ProB has state by state operation animation, and model checking Z using ProB ensures the correctness of the models, I have chosen Z as the final formal language and ProB as the final model checker for this research. Note that ProB now provides support to load Alloy models [2] by translating Alloy models into B machines (which is what Z specification translated into when it is loaded into ProB). This means our approach is possible to be further extended to support Alloy models in the future.

3.3 Plan of the research

In this section, I will discuss my plan for implementing the approach, including the consideration of the example and possible ways of achieving the approach.

At the start, I will construct a small example that is used to demonstrate our work. This example needs to be a safety-critical interactive system, the available options are medical systems such as infusion pumps, finance management system such as banking system, and transport system such as railway signalling and control system. To simplify the research process, the example should be small enough that makes it easy to explain and understand. As the example contains Z specifications and model checking takes a long time, the values selected in the example should also be small. The example should be expandable, so it could be used in a more generalised way by adding new features, which enables future work to be done.

The example should include a textual description that describes the functions of the system to be developed, as well as some Cucumber Scenarios that describe how they system should behave in different situations. These requirements should be presented in the perspective of business stakeholder, in order to simulate a typical software system development. Cucumber Scenarios can be written in many different languages. As this thesis is written in English, we will be using English for the language of the Cucumber Scenarios.

Apart from the requirements, the example should include a formal Z specification of the interactive system, specifying how the system should behave with given input, what the constraints of the behaviours are, and what the output should be after the operation. The Z specifications of the system will be model checked using ProB, in order to demonstrate the models' properties and operations.

Once the example is conducted, the implementation of the approach will start. Although Cucumber Framework was originally written in the Ruby programming language [46], the transformer will be implemented with Java, because I am more familiar with the Java version of the Cucumber library. As Cucumber Framework can run automated acceptance tests by hard-wiring the Cucumber Scenarios to the test implementation using step definitions, there might be some automated connection between Cucumber Scenarios and the acceptance tests. Thus the first possible way of implementing the approach is to use existing methods (if there are any) in the Cucumber Java library, to automatically transform (with modification) the Cucumber Scenarios into useful elements, such as constraints, predicates or direct models in any form, which can be modified or translated into Z specifications.

A code research will be done on the Cucumber Java library, to see if there are existing methods that can be used for automating Cucumber Scenarios to useful elements (predicates, or constraints or declarations) transformation. The elements are expected to specify the complete behaviours described in the Cucumber Scenarios, or match each step (the so-called step definition) of the behaviours described in the scenarios. As Cucumber Scenarios uses natural language, it is unlikely that the scenario can be directly translated into Z specification, that specify the behaviour described in the scenario. The Cucumber library we use is written in Java, this means these elements are likely to be directly readable by Java programs. Thus we only need to modify the elements into a form that can support the construction of Z specifications. The goal we are aiming to get is First-Order-Logic predicates. If we can get predicates from the Cucumber library, we will need to rewrite them into First-Order-Logic predicates (if they are not), so that a predicate can only refer to one single subject. This can be done by breaking the elements into small pieces that contains only one aspect of the behaviour, for example, the amount in the previously shown "Deposit Money" operation of the banking system, needs to

be positive. This deconstruction process can also help to preserve the information given in the original scenarios.

If no existing method is found that automatically translates Cucumber Scenarios into useful elements, we will have to implement the approach from scratch. We can start with reading the Cucumber Scenarios. A file reader can be used to read and store the steps in the scenarios. Keep in mind that Cucumber uses natural language which is very hard to generalise, we will need to either think of a way to limit the writing of scenarios, or omit some descriptive part that is not directly declaring the behaviour. As we want high correctness for supporting safety-critical interactive system development, we choose the former option, to avoid information loss during the transformation. Inspired by Colombo et al. [27], we therefore create rules to restrict the language used in the writing of Cucumber Scenarios. Remember that the Cucumber Scenarios are written by business stakeholders, thus the rules should be simple and easy for the stakeholders to understand. With the Cucumber Scenarios, we can translate each step in the scenario into First-Order-Logic predicate by manipulating the step text.

When we have the converter, we will experiment it on the small example we made and some general Cucumber Scenarios available from the Internet. The First-Order-Logic predicates converted from the example will be compared with the constraints in the Z specifications of the example, to see if there is information loss during the converting process. After that, we will experiment with constructing Z specifications using these predicates and model check them with ProB, to see whether there is error or not. This is for ensuring the correctness of the specifications. After finishing experimenting, we will evaluate the experiments we did during the research.

3.4 Summary

In this chapter we discussed the difference between Z notation and Alloy, and the difference between their analysers/model checkers. We explained the reason why we chose Z and ProB as the final tool for this research. The most important reason is that Alloy can't ensure the correctness due to its finite scope search while Z doesn't have this problem. We also described our plan on conducting the research, there are two considered ways of implementing the converter: The first way is to do a code research on the Cucumber Java library, to see if there is an existing method for transforming the Cucumber Scenarios. Then the output of such method (if it exists) is processed into First-Order-Logic predicates. The second way is to implement from scratch. Rules for writing the Cucumber Scenarios will be set and a converter will be implemented to convert each step in the scenario into First-Order-Logic predicates.

Chapter 4

Bank scenario example

4.1 Introduction

In this chapter, I will start by describing a simple Banking system which is an example of a safety-critical interactive system. Then I will describe how the banking system is developed in a typical software development process: starting with a business stakeholder giving out the requirements, then the requirements are translated into expected behaviours written as Cucumber Scenarios, then a formal Z specification declaring the functionalities of the desired system. Finally, we will demonstrate how to animate the Z specification in the ProB model checker and show related information.

4.2 Description

The example we use here is a simple banking system. Although medical systems and railway signalling and control systems are probably more directly re-

lated to human life, these systems are relatively domain-specific, which makes it harder for other readers without context knowledge to understand how these systems operate. Therefore we chose a banking system, as this safety-critical interactive system is what most people will use daily (even if people haven't used computer banking system before, they are likely to know about the operations in the bank), this makes the example easier for the readers to understand without having to learn anything. This banking system can be considered as an alpha version (so it has limited functionality) that allows swapping account type, making deposits, making withdrawals and gaining interest. The system is small with limited functionalities, but it contains all the necessary components that are needed for demonstrating the work in this thesis. An advantage of this example being small is that the functions are simple thus it will be easy to explain and understand, and the implementation will also be easier. As the considered functions are fairly independent, future work can be done by adding new features/functions to this example.

Assume we are in a real world business environment. We start off with the requirements from the business stakeholder responsible for requirements. From the bank operator's perspective, the main purposes are: keeping the money safe in the bank for the customers; Offer customers interest on deposits; Lending money to customers; Offering financial advice and related financial services. Some of these services are beyond the scope of our example, so they are omitted in what follows.
To summarise, we need a bank system that has different types of accounts, supports making deposits and withdrawals, and people can get interest on the money deposits. While keeping all these features, the bank system should be robust and safe.

With the requirements we gave above, we now describe a simple banking system that allows very basic operations such as deposit, withdrawal and earning

interest. These features are described as "DepositMoney", "WithdrawMoney" and "GainInterest" functions. The user can make a money deposit or withdrawal, which increases or decreases the amount of money in the account. Note, that there should be sufficient credit in the account for money withdrawal, the user can't withdraw more than the amount in the account. For safety purposes, a banking software should have a limit on the amount of money withdrawn or deposited in each operation, in order to reduce the loss if the account is stolen. We will limit this value to ($)6 (that is, the maximum amount in each withdrawal/deposit operation is 5) for demonstration purposes as 6 is enough for demonstrating our research, higher values will dramatically increase the amount of states/transitions to be checked, this will make the model checking process take a very long time (we discuss this further later). Based on the money in the account, the user can get interest regularly. To reduce the traffic, there should be some limit on the number of operations a customer can do per day, we set it to 6 (that is, the maximum number of operations that can be done by the account user per day is 5). We will need a function to reset this limit every day, this function will be "DayPass". In a real world banking environment, not all bank accounts can gain interest, to simulate this, we give an account type to each account which is initialised as "New", and the user can only gain interest if the account type is "Saving". The other account type is "Cheque" which can't do any extra operation due to the small scope of our example, but it can be further extended by adding operations only available for "Cheque" accounts. The user will have to set up the account type before doing any operation, this is achieved by "SwitchToSaving" and "SwitchToCheque" functions, these operations switch the user's account type to "Saving" or "Cheque" respectively. For the sake of model checking, we set up a limit for the credit that bank users can have in their accounts, if there is no limit, the model checking process won't finish. We set the credit limit to 21 (that is, the maximum amount of money in the account before or

after operations is 20). To conclude the limits we set up for the values, total credit of the account must be smaller than 21 (max value is 20), amount of money in withdrawal/deposit operations must be smaller than 6 (max value is 5), operations done everyday must be smaller than 6 (max value is 5). A new account should at least have some money in it, so we give the accounts ($)2 when they are initialised. With all the operations and limits introduced, we formed a small but complete bank system, we can use these requirements as a starting point for the Cucumber Scenarios we need later.

4.3 Cucumber Scenarios

Suppose we are the stakeholders, the textual requirements given above are not unambiguous enough to express our desired system, there could still be misunderstanding and confusion. We need behaviour specifications for the functions that the system is expected to have. These behaviour specifications will be given as Cucumber Scenarios. The Cucumber Scenarios are stored in Feature files, each "Feature" file is purely textual. The "Feature" file consists of a "Feature" which is a high-level description of one expected function of the system, and a "Scenario" (or a few scenarios) which is a concrete example that describes a situation. The "Scenario" illustrates the function with some user interactions and shows the results of these interactions. A "Scenario" consists of a few steps and each step starts with a keyword: "Given", "When", "Then", "And", or "But". "Given" describes the initial context, "When" describes the event and "Then" describes the expected outcome. "And" or "But" are optional, they are used to replace "Given", "When", "Then" if there are more than one of any of these three. The rest of the text in the scenarios is natural structured language such as English.

The Cucumber Scenarios have strong expressive power, an example of the

"Deposit Money" function can be described in a Cucumber Scenario as below (the text start with "#" symbol is the comment on the file):

```
Feature: Operation on the account
Banking operations such as depositing (adding) or
withdrawing (taking) money from an account.
Or switching account types such as from new to cheque
#Should do relevant effect to the account
#When operations today don't exceed 5
#Attempt to deposit money when operations today is smaller than 6
Scenario: Successfully deposit some money to an account
#Less than 6
Given Operationtoday is 4
And The current bank credit is 12
#amount < 6
#after operation, credit + amount = 16 < 21
When The amount is 4
Then add amount to credit
And add 1 to Operationtoday
#The credit should change
Then The credit should be 16
And Operationtoday should be 5
```

Figure 4.1: Cucumber feature for depositing money

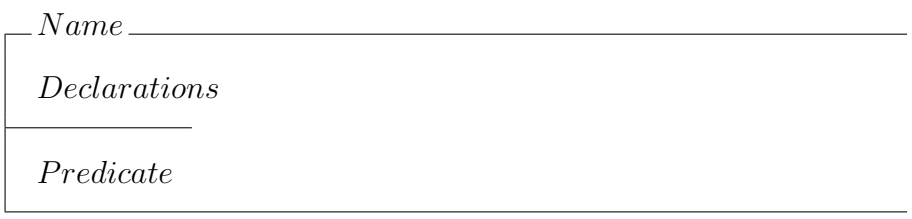
With the comments (text starts with "#" symbol) in Figure 4.1, we can see a Feature (this function could be separated as Deposit Money Feature or something later) for an operation on the bank account. The scenario is that given the number of operations done today is 4, it did not exceed 5, which means the user can still do more operations. The user has 12 dollars in the bank and attempts to deposit 4 dollars (which is smaller than the 5 dollars amount limit per operation). The operation is successful and the 4 dollars are added

into the user's account. The outcome of this scenario is that the current credit after the money deposit is 16 dollars and the operation done today count is increased to 5.

Here we demonstrated how the scenario works and how it can express a desired functionality of the system in the form of a behaviour. We can see that with the scenario, the making money deposit function is illustrated using natural language. This is particularly useful for giving requirements when the business stakeholder who is responsible for the requirements has no knowledge of coding or software development. The full Cucumber Scenarios for the banking example are given in Appendix A.

4.4 Z Specification

Next we consider a formal banking system example using Z notation.
When we write Z specifications, we aim at having a clear and formal description of the desired system in the form of a model. The goal of Z models is to support the production of the desired computer program written in some programming language. Bear in mind that the Z notation is a declarative language, we don't really care how exactly the program works, we only want to know the input, output and constraints, these are the declaration ("Account") schemas and the operation schemas ("DepositMoney", "WithdrawMoney", "DayPass" etc.). Thus the starting point of writing our Z specification, is to declare the attributes of the bank account. With a bank account, we can then create different operations schemas that consider different input, and give out the output respectively.
As we already discussed, the Z notation uses schemas to express the declarations and functions. Here is an example of what a schema looks like:



As shown above, the schema has a name, which expresses the purpose or the meaning of the functionality described in the schema. A schema has declarations in the top part which are the observations of state, they can be a list of declarations or reference to other schemas (called schema inclusion). If the schema is an operation schema, there will be a $\triangle$ symbol at the front of the name to indicate the state change. The bottom part of the schema contains predicates, these predicates could be constraints on the properties or how the properties should change in response to operations. The predicates are normally separated by new lines, the predicate part can be empty, which means the predicate part is assumed to be always "True".

In Figure 4.2 we show the Bank account schema written in Z, it is used as a declaration of the expected bank account:

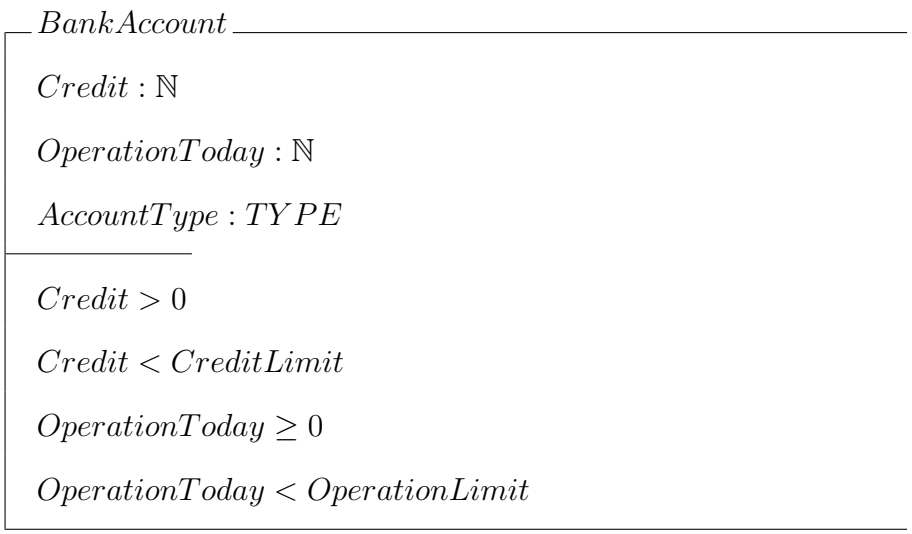


Figure 4.2: Bank account schema in Z

From Figure 4.2 we can see that a bank account has 3 observations (note here we used natural numbers for "Credit" and "OperationToday" for convenience purposes, it is enough to demonstrate our approach) : "Credit", which represents how much money the account holder has got in the account; "OperationToday", which represents how many operations the account has done today; "AccountType", which indicates what type the account is, it belongs to a preset set "TYPE" which contains "Cheque", "Saving" and "New". In the bottom part we can see the constraints on the variables: "Credit" must be greater than 0 because we assume an account should have money in it (also for convenience purpose). "OperationToday" must be greater or equal to 0, it makes no sense to have negative value of operations done, but "OperationToday" can be 0 when the account is initialised as a new account would not have done any operation. Note that "Credit" and "OperationToday" must be smaller than their limits as we discussed earlier.

In Figure 4.3 we show an example of the Deposit Money operation written in Z:

$$
\begin{array}{|l}
\hline
DepositMoney \\
\Delta BankAccount \\
Amount? : \mathbb{N} \\
\hline
\neg(AccountType = New) \\
Amount? > 0 \\
Amount? < AmountLimit \\
Credit' = Credit + Amount? \\
OperationToday' = OperationToday + 1 \\
AccountType' = AccountType \\
\hline
\end{array}
$$

Figure 4.3: Deposit Money operation in Z

In Figure 4.3 can see that there is a BankAccount schema included with $\triangle$ indicating it is a state change schema, and an "Amount?" (the "?" indicates this is an input to the operation) value for the money to be deposited which is a natural number. In the predicate part, we can see the requirements we talked about earlier fulfilled: A new account can't do this operation ($\neg$ (AccountType = New)); The money to be deposited can't exceed the AmountLimit (it is set to be 6). After the state change, the operation today will be the operation today before state change plus 1, this is indicated by the "OperationToday' = OperationToday +1". New credit in the account will be the old credit plus the money to be deposited and the Account type will remain the same.

Here we showed the structure of Z schemas and how it can be used to declare a function/operation of the model of the system desired. The mathematical syntax might be hard to understand if the stakeholder without related knowledge tries to read it, but the Z models are not aimed at stakeholders so we don't need to worry about that. As we can see, the schema demonstrated how the system changes from one state to another, giving a formal and unambiguous way to describe the expected behaviour of the desired system. The full Z specification for the banking example is given in Appendix B.

4.5 ProB (ProZ)

As we discussed, ProZ is a model checker that allows a model to be animated and checked. In this section I will demonstrate loading the Z notation we provided in ProZ, and animate the model to show some operations it enables.
In Figure 4.4 the example of the Z model loaded in ProB after initialisation is shown.

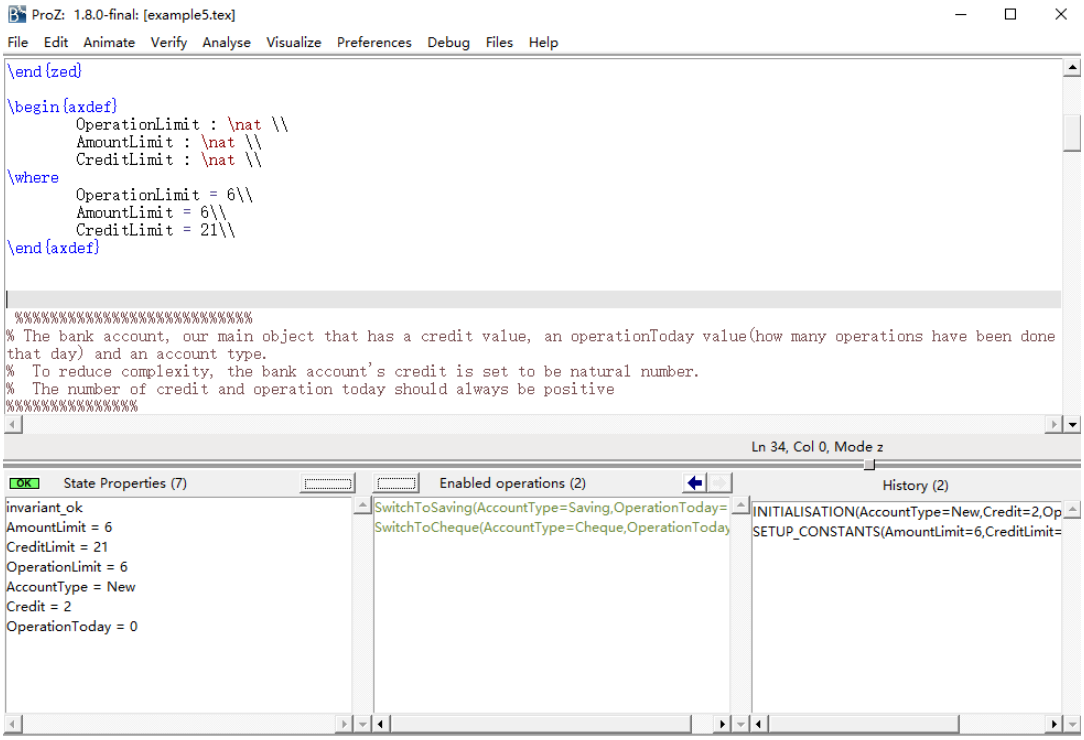


Figure 4.4: Loaded and initialised bank model in ProB

As shown in Figure 4.4, we have setup constants and finished initialisation (in the History section on the bottom right of the figure). In the figure we can see the current state properties, as we explained earlier, the Amount limit is 6, the Operation limit is 6 and the Credit limit is 21. The account has just been initialised, thus its account type is New, which means it can't do any operation before setting up the account type. The "OperationToday" is 0 as there is no actual operation done (initialisation and setup constants don't count as operations done on the account in our case). In the middle bottom part we can see the enabled operations, we can choose either "SwitchToSaving" or SwitchToCheque, to demonstrate the "GainInterest" function, we will switch the account type to "Saving".

In figure 4.5 we showed the state of the Z model after switching account type to "Saving":

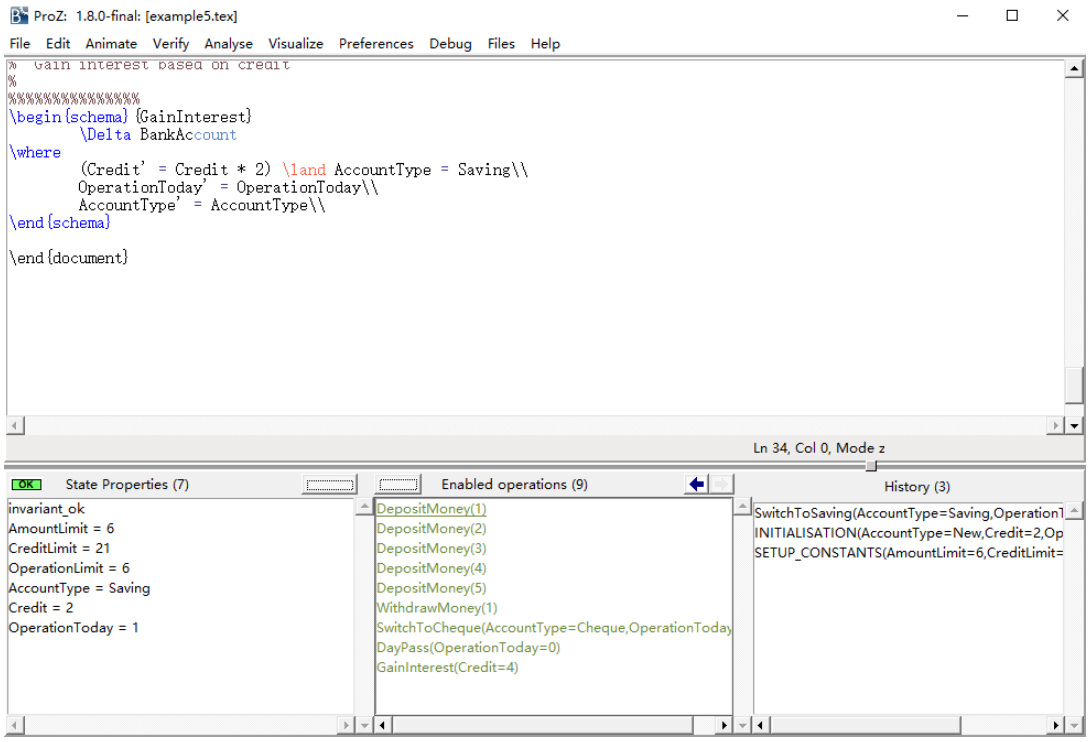


Figure 4.5: Bank model in ProB after switching to "Saving" account type

As shown in Figure 4.5, we can see that the "AccountType" of current state is now "Saving", this enabled all the operations we described in the previous sections, including "DepositMoney", "WithdrawMoney", "SwitchToCheque", "DayPass" and "GainInterest". These operations will change the state properties respectively. Note that the "DepositMoney" operation can only deposit up to ($)5 into the account, this is because the amount must be smaller than 6. The "WithdrawMoney" can only withdraw ($)1, because the credit is only ($)2 in the account and the amount must be smaller than current Credit.

Next, we demonstrate "GainInterest" function of the banking example:

In the Figure 4.6 we can see the schema of "GainInterest" function, the "Credit" of the account will be doubled (this is a high interest rate but we keep it 2 for the sake of simplification) and the OperationToday will be increased by 1.

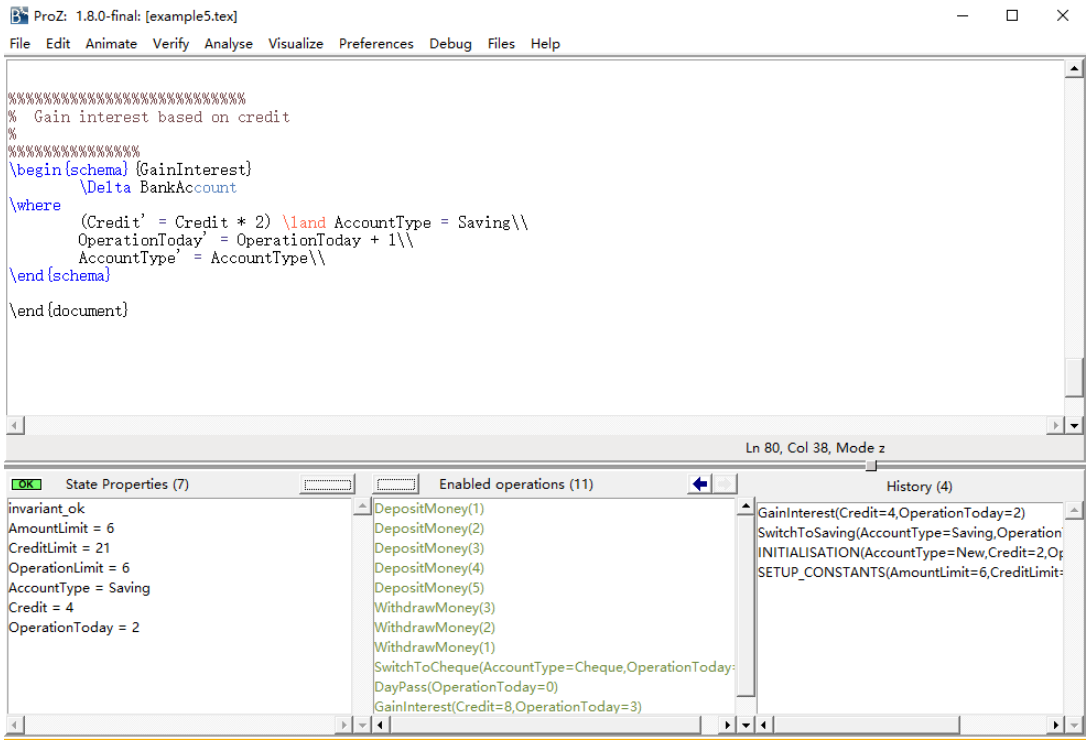


Figure 4.6: Bank model in ProB after gaining interest

In the State Properties section, we can see that the current "Credit" has been correctly increased to 4 and "OperationToday" has been increased to 2. As "DepositMoney", "WithdrawMoney" and "DayPass" functions are simply addition and subtraction on the values, we won't demonstrate the actual use of these functions. Next, we will show how the Z model is checked.

The position of the model check function in ProB is shown in Figure 4.7, now we will check the bank model by clicking on the highlighted button.

As shown in Figure 4.8, we have checked the bank model. Note that expanding the model states can be recursive, the model checker could possibly generate infinite states if we don't limit the number of recursions, we use default ProB settings for the maximum number of initialisations and the maximum number of enablings per Operation (due to the size of our example being small, all the distinctive states can be checked). By conducting the check, ProB has checked all distinct states and transitions, no error state (deadlocks etc.) was found. Note that we did not set up any invariant so there won't be any invariant

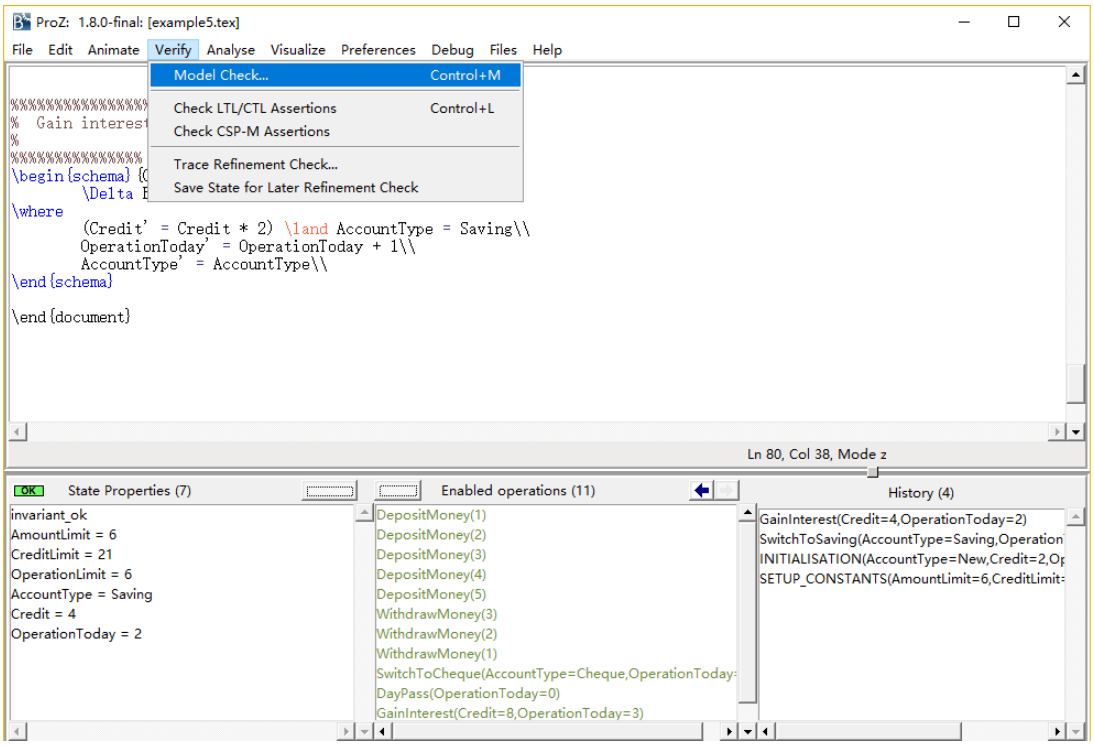


Figure 4.7: Model check function in ProB

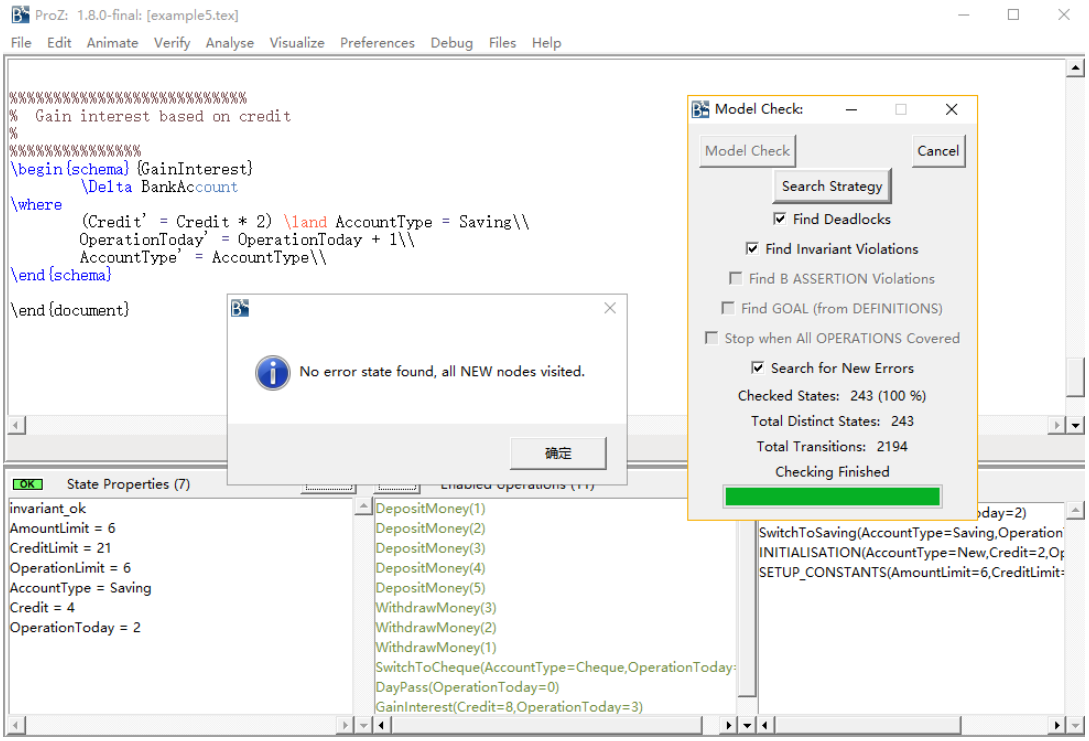


Figure 4.8: Result of model checking the bank model in ProB

violations. The result implies that for the requirements we implemented in our specification, there is no erroneous state in the model that will lead to a

deadlock or system crash, that is, the system is correct.

4.6 Summary

In this chapter, we explained the small banking example we made for the research, including the textual system requirement description and Cucumber Scenarios written from the perspective of the business stakeholder. We also explained the Z specification we made for this banking example and loaded it in the ProB model checker, along with some demonstration on what could be done with ProB model checker.

Chapter 5

Implementation of The Approach

5.1 Introduction

In this section, I will start by discussing the challenges of transforming Cucumber Scenarios to First Order Logic Predicates and describing initial experiments. Then I will talk about the converter and explain how it evolved. Next, I will demonstrate a use of the converter with the bank example discussed earlier as a case study. Then I will discuss testing the converter with other BDD specifications (Cucumber Scenarios). Finally, I will explain how the predicates can be transformed into Z predicates.

5.2 Initial experiments

The first step was to consider how to convert raw Cucumber Scenarios to some elements that can be used for further transformation. Cucumber Scenarios are written in plain language that consumers can read and write. Thus it can suffer

from the problem when we are attempting to automatically convert Cucumber Scenarios into First-Order Logic - the meaning of the text heavily relies on the grammar the user is using and the context. (note that Cucumber supports over 70 languages but we will only focus on English in our research.)
Since Cucumber automatically combines the Cucumber Scenarios and the Cucumber test glue code (for example Java code that defines the tests), we started with the assumption that there might be some transformations that we can use in the existing Cucumber library.

Now we look at an example of the Cucumber test glue code for step "Given OperationToday is " (\\d+)$" (it means multiple digits)" (note that this test method is a general transformation from a step in the original scenario):

```
@Given ("^Operation today is (\\d+)\$")
public void operation_today_is (int newOperToday) throws Exception {
// Write code here that turns the phrase above into concrete actions
this.operToday = newOperToday;
}
```

We can see that the Cucumber library automatically connects the steps in the form of "Given OperationToday is " (\\d+)$" (multiple digits)" in the scenarios to the method that has metadata:

```
@Given ("^Operation today is (\\d+)\$")
```

This made us wonder if there is some transformation that can turn Cucumber steps into predicates or other constraints/declarations, which can be matched with the Java glue code. If we found the transformation method, it could be very useful. Since we know the input is a Cucumber step and the output is likely to be predicates, we can try different methods on the library. We can directly use the transformation method and transform the output to be the

predicates we can use in Z specification. Or we can take out the method that does the transformation as a base and modify it to output predicates we can directly use in Z specification.

As we planned, a code research was done on the library in order to find the transformation method. Unfortunately, we didn't find such a method that does any actual transformation on the steps. In the Cucumber library, the Cucumber test glue code was grouped and processed into patterns. The Cucumber steps in the given Cucumber Scenarios are matched with the patterns from glue code, by comparing the step text and the text in the metadata, if they match, the corresponding method for this step is found. This method we found in the Cucumber Java library is not exactly what we are looking for, so we have to find another way to achieve our goal.

There were approaches on automatically translating natural language sentences into First-Order-Logic. Baral et al. proposed the Inverse Lambda Calculus [15] algorithm with the help of Combinatory Categorial Grammar (CCG) [26]. The downside of Baral's approach is the approach heavily relies on CCG, this is similar to the problem we stated, the meaning of the text heavily relies on the grammar the user is using. But we learnt from Baral's approach, if the grammar gives the text different meaning, we can limit the grammar that the writer of the natural language sentences use.

Cucumber Scenarios follow the Gherkin grammar rules (or we can call it BDD grammar, unambiguous behaviour specifications that describe how the software should behave). It is a downside for transformation from plain English to First-Order Logic predicates but we can also use it to restrict how the scenario writers (such as business stakeholders) manage their way of expressing requirements.

In the manner of Colombo et al. [27], we set up some rules for how the scenario writer describe the expected behaviours (these are the final rules we got after conducting the experiments):
When a stakeholder writes a Cucumber step (starting with a key word such as Given), the step must be from one of the 3 groups. The first group is declaration, if we want to state that A belongs to or is equal to B, or A is in the state of B, the step should be written in the form of "A is/are/should be B", depending on whether A is plural or singular. For example:

```
Given The current bank credit is 15
```

The second group is calculation, only simple algebra calculation is allowed in the steps. The calculation can be addition, in the form of "add A to B", or subtraction, in the form of "subtract A from B". For example:

```
And add amount to credit
```

The third group is action, if we want to describe an action is done on object B by subject A, the step should be written in the form of "A makes/make/do/-does on/to/from/by/with B", depending on whether A is plural or singular, and what the action is. There can only be one object and one subject. For example:

```
Then Paul does switch accountType to cheque
```

While writing the steps based on the three groups, each step should only have one statement. By following the rules to write Cucumber steps, we are able to simplify and limit the meaning of the step. Therefore the difficulty of converting the step into predicates is reduced.

5.3 Implementing the converter

In this section I will talk about the three versions of the converter and how I implemented them. The converter is a text processer, it will take a Cucumber feature file as argument. The feature file will be read and processed, each step of each Cucumber Scenario will be transformed into a First-Order-Logic predicate, thus each Cucumber Scenario will be converted into a group of First-Order-Logic predicates describing the behaviour described in the scenario.

5.3.1 Initial version of converter

The first approach was based upon simple string conversions. We start by looking at the small Cucumber Scenario used for testing the converter in Figure 5.1:

```
Feature: Operation on the account
Banking operations such as depositing (adding) or
withdrawing (taking) money from an account.
Or switching account types such as from new to cheque

#Attempt to deposit money when operations today don't exceed 5
Scenario: Deposit some money to an account
Given Operation today is 4
And The current bank credit is 15
When The amount is 5
Then User's credit is added by amount
And Operation today is added by 1
#The credit should change
Then The credit should be 20
```

```
And operation today should be 5
```

Figure 5.1: Successfully Deposit Money Scenario

This example shown in Figure 5.1 contains only calculation and declaration, which were all that the initial version of the converter could handle. Based on this example, we want to transform the steps into some sort of declaration predicate that describes an attribute of the current state, for example, imagine we loaded a Z specifcation into ProB:

```
Given Operation today is 4
```

This step should convert into "Operation today = 4", declaring the current value of OperationToday is 4.

The first step of the transformation is to convert keywords of the steps, each keyword (Given, When, Then, And) of the steps in the scenarios is directly transformed to "E", "A", "Imp", "A" respectively. "E" means exists $\exists$, "A" means logical conjunction $\wedge$ and "Imp" means implication $\Rightarrow$. Note that only the first "Then" detected will be transformed into Imp, the other "Then" keywords will be transformed into A (because the steps after the first "Then" are all outcomes of the scenario).

Here is the pseudo code for transforming the keyword:

```
if keyword is "Given" then
    print "E";
else if keyword is "When" then
    print "A";
else if keyword is "Then" then
    if First "Then" indicator = True;
     then
        print "Imp";
        First "Then" indicator = False;
    else
        print "A";
    end
else if keyword is "And" then
    print "A";
```

Algorithm 1: Transforming the keyword

By checking the keyword of each step, I was able to turn on/off the indicator for the "Then" keyword. This indicator indicates whether a new scenario has started and the first "Then" detected in the new scenario should be converted to "Imp". If another new scenario started after the indicator is turned off, it will be turn on again. The keywords are also used to indicate that a new step is detected, this can be used to skip comments and as a endpoint for the previous step.

The first working version could only handle simple calculations such as equality (A = B), addition (A + B) and subtraction (A - B). The words are simply appended at the end of each other until an indicator is detected. The indicators are "is" and "should". When "is" is detected, the word after is will be checked. If "is" is followed by "added", the action of this step will be grouped as "addition". If "is" is followed by "subtracted", the action will be grouped

as "subtraction". If indicator "is" is not followed by these two, there will be some other check and this step will be grouped as "equality".

A step can then be transformed into a predicate depending on what group it is in and what calculation it is doing. For example, "And Operation today is added by 1" will be grouped as "addition", then the step will be translated as "A Operation today = Operation today + 1".

The complete pseudo code of the converter version 1 is shown in Algorithm 2.

The scenarios I tried to transform with the Version 1 converter are shown in Figure 5.2:

```
Feature: Operation on the account
Banking operations such as depositing (adding) or
withdrawing (taking) money from an account.
Or switching account types such as from new to cheque
Scenario: Deposit some money to an account
Given Operation today is 4
And The current bank credit is 15
When The amount is 5
Then User's credit is added by amount
And Operation today is added by 1
Then The credit should be 20
And operation today should be 5
Scenario: Unsuccessfully deposit some money to
an account with too high operationsToday
Given Operationtoday is 5
And  The current bank credit is 15
When The amount is 4
Then The credit is not changed
Then The credit should be 15 and operationtoday should be 5
```

Figure 5.2: Scenarios tried with Version 1 Converter

Data: Cucumber feature file

Result: First-Order-Predicate

```
1  load the feature file;
2  read the feature file line by line;
3  while not at end of this feature file do
4      read the line word by word separated by " " (white space);
5      if First word is "scenario" then
6          First "Then" indicator = True;
7      else if First word is a keyword then
8          transform the keyword;
9          keep reading the line;
10         while not at end of this line do
11             if Word is "is" then
12                 if "is" is followed by "added" then
13                     step is addition;
14                 else if "is" is followed by "subtracted" then
15                     step is subtraction;
16                 else
17                     step is declaration;
18                 end
19             else if Word is "should" then
20                 step is declaration;
21         end
22         print the predicate based on the group of the step
23     else
24         print the line;
25     end
26 end
```

Algorithm 2: Transforming a step into First Order Predicate

Note that in the second scenario in Figure 5.2, the last step has an extra "and", this was because I didn't set any scenario writing convention when I was working on the first version. This extra "and" did lead to problems that the last step is not converted into a First-Order-Logic predicate.

```
Scenario: Deposit some money to an account
E Operation today = 4
A The current bank credit = 15
A The amount = 5
Imp User's credit = User's credit + amount
A Operation today = Operation today + 1
A The credit = 20
A operation today = 5


Scenario: Unsuccessfully deposit some money to an account with too high operationsToday
E Operationtoday = 5
A  The current bank credit = 15
A The amount = 4
Imp The credit = The credit
A The credit = 15andoperationtodayshouldbe5
```

Figure 5.3: Output of converter version 1

The output from the converter is shown in Figure 5.3, we can see that the last step of the second scenario has a long string appended at the end, this was because each line read only checked the keyword once.

Using the scenario "Deposit some money to an account" as example, we can see "Given Operation today is 4" was mapped into "E Operation today = 4", showing that, in the scenario, the account has done 4 operations today. The output of transforming the unsuccessful money deposit scenario is equivalent to the following First-Order-Predicates: $(\exists Operationtoday = 5 \wedge credit = 15 \wedge amount = 4) \Rightarrow (credit = credit \wedge (credit = 15 and operationtodayshouldbe 5))$.

The initial version could only handle simple addition/subtraction in the form of A is added to B/C is subtracted by D. Converter version 1 couldn't handle any real actions such as "John inserted a coin". From the examples above we found that it does not handle extra keywords properly. My solution to the extra keywords was setting up scenario writing rules so there can only be one

keyword and action in each line (which means one statement per step). The actions were difficult to parse, but the powerful thing about First-Order Logic is that we don't need to care how exactly the action will executed, we only care about the input and output. This led to the second version of the converter.

5.3.2 Converter Version 2

The problem of the converter version 1 is that it could not transform any action or multiple keywords in the same step. Thus one goal of converter version 2 is to differentiate actions (A does B) and declaration (A is B), the other goal is to somehow recognise and deal with more than one keywords in each step. In this version, steps are categorised as "Action", "Calculation" or "Declaration" in order to represent general actions. As with converter version 1, keywords are converted and used as starting point of each line. There are 4 rules added for writing the scenarios: Each line can only have 1 keyword; Each step can only have 1 object and 1 subject; The object/subjects should be a single word (for example, "account type" should be written as "accounttype" or "account-Type"); Each step must be in the form of each category.

For the categories, action steps should be in the form of A do/does/-make/makes something to/from/on/by B, which will be converted to something(A,B). Calculation steps should be in the form of add/subtract A from/to B, which will be converted to add(A,B) or subtract(A,B). Declaration steps should be in the form of A is/should be/are B or A is not changed, which will be converted to A = B or A = A.

When we were deciding the rules, we realised that the typical way of declaring A is B in First-Order Logic predicates is "isB(A)", for example, "Sky is blue" will be expressed as "isblue(Sky)". But in our case, we want to keep the ability of describing numbers, such as "Credit = 5", which is very useful in Z specifications. Potentially the approach can be improved by dealing with

adjectives in a smarter way in the manner of "isB(A)" while keeping the ability of describing numbers.

The following scenarios were used as examples of transformations with the Converter Version 2:

```
Feature: Operation on the account
Banking operations such as depositing (adding) or
withdrawing (taking) money from an account.
Or switching account types such as from new to cheque

Scenario: Unsuccessfully withdraw some money
from account with insufficient credit
Given Operationtoday is 4
And The current bank credit is 4
When The amount is 5
Then The credit is not changed
Then The credit should be 4
And Operationtoday should be 4

Scenario: Successfully switch accounttype from new to cheque
Given AccountType is new
And Operationtoday is 0
And Paul does switch AccountType to cheque
Then add 1 to Operationtoday
Then Accounttype should be cheque
And Operationtoday should be 1
```

Figure 5.4: Scenarios tried with Version 2 Converter

The result of transforming the Cucumber Scenarios in Figure 5.4 is shown in Figure 5.5:

```
Scenario: Unsuccessfully withdraw some money from account with insufficient credit
E Operationtoday = 4
A The current bank credit = 4
A The amount = 5
Imp The credit = The credit
A The credit = 4
A Operationtoday = 4


Scenario: Successfully switch accounttype from new to cheque
E Accounttype = new
A Operationtoday = 0
A switchAccountType(Paul,cheque)
Imp add(1,Operationtoday)
A Accounttype = cheque
A Operationtoday = 1
```

Figure 5.5: Output of converter version 2

From Figure 5.5, we can see that the output of converting "Successfully switch accounttype from new to cheque" scenario using converter version 2 is equivalent to the following First-Order-Predicates: $(\exists Accounttype = new \wedge Operationtoday = 0 \wedge switchAccountType(Paul, cheque)) \Rightarrow (add(1, Operationtoday) \wedge Accounttype = cheque \wedge Operationtoday = 1)$, the simple actions following the writing rules are successfully converted.

This version managed to transform simple actions, but it didn't consider the situation where A is not B or "A'" (which indicates the before and after state change in Z specification). To address these problems, version 3 of the converter was developed.

5.3.3 Converter Version 3

In the third version, the "A is not B" situation is solved by checking the next two words of the step. Since the scenario is written by human stakeholders, we can set up a rule that a step must be meaningful. Therefore a step such as "A is not changed" where changed is the name of "B", is not allowed. By

eliminating this possibility, we can simply check the word after "is not", if the word is "changed", we consider the step as "A = A", otherwise the step will be considered as "A != B" ($A \neq B$).

For checking the before and after states of the variables, we considered the "Then" keyword. A Cucumber Scenario must contain at least one "Then" keyword as the consequence or result of the scenario, thus we can make use of "Then" to recognise the before state predicates and after state predicates. The steps before "Then" will be the predicates before state change, and the steps after "Then" will be the predicates after state change. Note that there can be multiple occurrences of the "Then" keyword in a scenario, here we made use of the "First 'Then' Indicator". When the value of the "First 'Then' Indicator" is still True, we haven't encountered the first "Then" yet, which means all steps we read were before state change. Once we encounter the first "Then", the value of the indicator becomes False, which means all the steps we read after this step are after state change.

Knowing whether before or after state a variable is in, if the variable occurred in both before and after states, we can find the primed version of this variable in the after state. By using this feature, we need to make sure that the stakeholder who writes the scenarios is consistent with the names, capitalisation must be the same too.

The following scenarios were used as examples of transformations with the converter Version 3:

```
Feature: Operation on the account
Banking operations such as depositing (adding) or
withdrawing (taking) money from an account.
Or switching account types such as from new to cheque

Scenario: Unsuccessfully withdraw some money
from account with insufficient credit
```

```
Given Operationtoday is 4
And current bank credit is 4
And The current bank credit is not 5
When The amount is 5
Then The credit is not changed
And credit should be 4
And Operationtoday should be 4

Scenario: Successfully switch Accounttype from new to cheque
Given Accounttype is new
And Operationtoday is 0
And Paul does switch AccountType to cheque
Then add 1 to Operationtoday
Then Accounttype should be cheque
And Operationtoday should be 1
```

Figure 5.6: Scenarios tried with Version 3 Converter

The result of transforming the Cucumber Scenarios in Figure 5.6 is shown in Figure 5.7:

```
Scenario: Unsuccessfully withdraw some money from account with insufficient credit
E Operationtoday = 4
A credit = 4
A credit != 5
A amount = 5
Imp credit' = credit
A credit' = 4
A Operationtoday' = 4


Scenario: Successfully switch accounttype from new to cheque
E Accounttype = new
A Operationtoday = 0
A switchAccountType(Paul,cheque)
Imp add(1,Operationtoday)
A Accounttype' = cheque
A Operationtoday' = 1
```

Figure 5.7: Output of converter version 3

In Figure 5.7, the output of unsuccessful money withdrawal is equivalent to the following First-Order-Predicates: $(\exists Operationtoday = 4 \wedge credit = 4 \wedge \neg(credit = 5) \wedge amount = 5) \Rightarrow (credit' = credit \wedge credit' = 4 \wedge Operationtoday' = 4)$. We can see that "A is not B" situation is solved and the primed variables in the after states are addressed.

In the after states, only the left hand side of the equation is appended with a prime symbol, indicating the variable in this step is the version after the operation. Note that only the words before "is", "should" and "does" are recorded as the variable names, as we discussed earlier, the object/subject in the step should be a single word, this rule helps the transformation to get simpler variable names.

In the converter version 3, existing problems are solved, we will use this as the final version, there are still many aspects (such as it can't handle descriptive adjectives that are not directly describing values, for example, A is drinking water, in a typical First-Order-Logic manner this sentence should be converted into is_drinking(A, water), but in our converter this sentence will be converted into A = drinking water) to be improved but they are beyond the scope of this research. Next, we will test the converter with some Cucumber Scenarios.

5.4 Testing with bank scenarios

In this section we will apply the converter version 3 to the bank system scenarios we made (in Appendix A). We will start with modifying the Cucumber Scenarios to meet our new grammar rules described in the previous sections and put them in a feature file (which is what the program reads as input), then the feature file will be processed by the converter, the output will be a text file containing the predicates converted from the Cucumber steps. After transforming, we will check if there is information lost.

Here we take a money deposit scenario from the banking system example that contains action as an example:

```
1  Feature: Operation on the account
2  Banking operations such as depositing (adding) or
3  withdrawing (taking) money from an account.
4  Or switching account types such as from new to cheque
5  #Attempt to deposit money when operations today don't exceed 5
6  Scenario: Successfully deposit some money to an account
7  #Less than 5
8  Given Operationtoday is 4
9  And The current bank credit is 15
10 When The amount is 5
11 Then add amount to credit
12 And add 1 to operationtoday
13 #The credit should change
14 Then The credit should be 20
15 And operationtoday should be 5
```

Figure 5.8: Money Deposit Scenario

In Figure 5.8 given above we can see that variables "Operationtoday" and "credit" occurred before and after the first "Then" step, which means these two variables need to be primed in the after state. Note that the first "Operationtoday" in line 8 starts with upper case but the other "operationtoday" in line 12 and 15 start with lower case, this is against our rule, we need to modify them so all of them are exactly the same. There are also comments which slows down the reading process, but we don't remove them as we want to keep the information in the Cucumber and simplify the modifying process. The rest of the steps meet our rules so we don't have to adjust them.

Here is the scenario after the modification:

```
1  Feature: Operation on the account
```

```
Banking operations such as depositing (adding) or
withdrawing (taking) money from an account.
Or switching account types such as from new to cheque
#Attempt to deposit money when operations today don't exceed 5
Scenario: Successfully deposit some money to an account
#Less than 5
Given Operationtoday is 4
And The current bank credit is 15
When The amount is 5
Then add amount to credit
And add 1 to Operationtoday
#The credit should change
Then The credit should be 20
And Operationtoday should be 5
```

Figure 5.9: Money Deposit Scenario After Modification

Now the scenario in Figure 5.9 is ready to be put into the converter, the output is shown in Figure 5.10.

```
#Attempt to deposit money when operations today don't exceed 5
Scenario: Successfully deposit some money to an account
#Less than 5
E Operationtoday = 4
A credit = 15
A amount = 5
Imp add(amount,credit)
A add(1,Operationtoday)
#The credit should change
A credit' = 20
A Operationtoday' = 5
```

Figure 5.10: Output of testing modified bank scenario

The output is equivalent to the following First-Order-Predicates:
$(\exists Operationtoday = 4 \wedge credit = 15 \wedge amount = 5) \Rightarrow (add(amount, credit) \wedge add(1, Operationtoday) \wedge credit' = 20 \wedge Operationtoday' = 5)$ As we expected, only the single word variable names are kept in the predicates, the comments

are printed without changing and the after state variables are primed correctly. The result is correct and the main information given in the scenario is transformed into predicates without losing any information. After repeating the same process, the whole banking system scenario was transformed without error.

5.5 Testing with other scenarios

In this section, we will take one example of an arbitrary scenario and apply the same process on the example to transform into first-order logic predicates. This example was taken from [10], the point of testing this example is to check that our converter works in cases where features weren't written by me. This example does not contain many adjectives (which suits our converter), and the feature is also describing a bank operation while the writer is not me, which makes it suitable for our demonstration. We will check if there is information lost after the transformation. Here is the bank operation example [10]:

```
1  Feature: Account Holder withdraws cash
2
3  Scenario: Account has sufficient funds
4  Given the accountBalance is $100
5  And the card is valid
6  And the machine contains enough money
7  When the Account Holder requests $20
8  Then the ATM should dispense $20
9  And the account balance should be $80
10 And the card should be returned
```

Figure 5.11: Arbitrary Scenario

We can see that the structure of the example shown in Figure 5.11 is similar to

the scenarios we made. Now we modify the scenario to match our previously defined rules and see if there is information lost.

```
1  Feature: Account Holder withdraws cash
2
3  Scenario: Account has sufficient funds
4  Given the accountBalance is $100
5  And the card is valid
6  And the machine does contain with enoughMoney
7  When the AccountHolder does request with $20
8  Then the ATM does dispense with $20
9  And the accountBalance should be $80
10 And the card should be returned
```

Figure 5.12: Arbitrary Scenario After Modification

In the modified scenario shown in Figure 5.12 we can see that there is not much text modified, only the structure of the steps is slightly different, thus we are confident that there is no information lost in the modifying process. Now we put the modified scenario into the converter and see what the result looks like:

```
Feature: Account Holder withdraws cash

Scenario: Account has sufficient funds
E accountBalance = $100
A card = valid
A contain(machine,enoughMoney)
A request(AccountHolder,$20)
Imp dispense(ATM,$20)
A accountBalance' = $80
A card' = returned
```

Figure 5.13: Output of testing modified arbitrary scenario

In Figure 5.13 the output is equivalent to the following First-Order-Predicates: $(\exists accountBalance = \$100 \wedge card = valid \wedge contain(machine, enoughMoney) \wedge request(AccountHolder, \$20)) \Rightarrow (dispense(ATM, \$20) \wedge accountBalanace' =$

$\$80 \wedge card' = returned$). We can see that the actions done in the scenario are represented in the predicates, the values of the variables before and after the actions are correctly transformed. Some actions might be slightly unintuitive to human readers, but the actual meaning of the actions should be easy to understand. Comparing the predicates with the original scenario, we are confident that there is no information lost in this transformation.

5.6 Using predicates to build Z specifications

After testing the converter with 10 different Cucumber Scenarios available on the Internet [4, 5] and got reasonably accurate results, we are confident that the predicates from transforming the scenarios maintained the information of the requirements that the writer of the scenarios was trying to give. Now we try to use the predicates transformed from the banking system example we gave to compare the predicates with the Z specifications, in order to assist with the development of a Z specification model.

Now we look at the money deposit scenario and the predicates created by the converter.

```
1  Feature: Operation on the account
2  Banking operations such as depositing (adding) or
3  withdrawing (taking) money from an account.
4  Or switching account types such as from new to cheque
5  #Attempt to deposit money when operations today don't exceed 5
6  Scenario: Successfully deposit some money to an account
7  #Less than 5
8  Given Operationtoday is 4
9  And The current bank credit is 15
10 When The amount is 5
```

```
11 Then add amount to credit
12 And add 1 to operationtoday
13 #The credit should change
14 Then The credit should be 20
15 And operationtoday should be 5
```

Figure 5.14: Bank Scenario After Modification

```
#Attempt to deposit money when operations today don't exceed 5
Scenario: Successfully deposit some money to an account
#Less than 5
E Operationtoday = 4
A credit = 15
A amount = 5
Imp add(amount,credit)
A add(1,Operationtoday)
#The credit should change
A credit' = 20
A Operationtoday' = 5
```

Figure 5.15: Output of testing modified bank scenario

The output of transforming the bank scenario in Figure 5.14 is shown in Figure 5.15. As we explained in previous section, the output is equivalent to the following First-Order-Predicates:

$(\exists Operationtoday = 4 \wedge credit = 15 \wedge amount = 5) \Rightarrow (add(amount, credit) \wedge add(1, Operationtoday) \wedge credit' = 20 \wedge Operationtoday' = 5)$

It is clear that this scenario can be used to form a money deposit function. This function can be described using an operation schema. The name of this schema is "Money deposit". From the scenario description:

```
1 Feature: Operation on the account
2 Banking operations such as depositing money
```

It is obvious that the operation was done on the (bank) account, thus we get a predicate for the declaration part of the operation schema: $\triangle$account

Looking at the predicates we got from the output, there are two actions: add(1,Operationtoday) and add(amount,credit). 1 is a constant number, "amount"

only appeared in before state, while "Operationtoday" and "credit" appeared in both before and after states. Only "amount" can be used as the input. As we can only see amount is a positive integer from "amount = 12", we get Amount?:ℕ for the declaration part of the schema, which means Amount? belongs to the natural numbers set.

The predicates "add(amount,credit)", "add(1,Operationtoday)" can be directly used in the function part of the schema. Therefore we formed an operation schema:

$$
\begin{array}{|l}
\hline
Moneydeposit \\
\Delta account \\
amount? : \mathbb{N} \\
\hline
add(amount, credit) \\
add(1, Operationtoday) \\
\hline
\end{array}
$$

With the primed observations, we can see that their new values are the results of the actions. Thus we get:

$$
\begin{array}{|l}
\hline
Moneydeposit \\
\Delta account \\
Amount? : \mathbb{N} \\
\hline
credit' = add(Amount?, credit) \\
Operationtoday' = add(1, Operationtoday) \\
\hline
\end{array}
$$

This schema is built with the information we gained, based on the predicates output of the original Cucumber Scenarios. Now we compare the schema with the DepositMoney schema we gave earlier.

$$
\begin{array}{|l}
\hline
DepositMoney \\
\hline
\Delta BankAccount \\
Amount? : \mathbb{N} \\
\hline
\neg(AccountType = New) \\
Amount? > 0 \\
Amount? < AmountLimit \\
Credit' = Credit + Amount? \\
OperationToday' = OperationToday + 1 \\
AccountType' = AccountType \\
\hline
\end{array}
$$

We can see that the declaration parts are almost exactly the same. The predicate part of the "Moneydeposit" schema is smaller than the function of "DepositMoney" schema but "DepositMoney" schema has all the predicates in the "Moneydeposit" schema. This is because we were only building the schema based on one scenario, normally there should be multiple scenarios describing the same function under different circumstances. Given the fact that everything in "MoneyDeposit" schema is in the "DepositMoney" schema, we are sure these two schemas are consistent, both schemas supports the scenario above. With all the scenarios we should be able to construct a complete schema that is sufficient to describe the whole function.

As the predicates are close to a subset of the separated Z notation schemas, we can‘t model check the new schema without modifying the new schema variable names. But here we get the idea that the approach can be further improved by allowing automatic model checking.

5.7 Evaluation

Based on the tests we did, we are confident that our approach can successfully translate simple Cucumber Scenarios into useful First-Order-Logic predicates, while maintaining the main information of the scenarios. We have shown that these predicates from the Cucumber Scenarios can be used to create a partial Z schema, which belongs to the independent Z specification that specify the same banking system, that the Cucumber Scenarios are describing. This implies that if we have a complete (enough) set of Cucumber Scenarios describing a safety-critical interactive system, we can form a full set of First-Order-Logic predicates, which can be used to support the construction of a formal Z specification for this system. When a business stakeholder gives out the requirements of a safety-critical system using Cucumber Scenarios, this work could transform the scenarios into First-Order-Logic predicates, which can be used to check with the formal Z specifications constructed by the developers, to see if the Z specifications meet all the requirements given in the scenarios.

Chapter 6

Conclusion

6.1 Introduction

In this chapter, we will start with reviewing the goal of our research. Then we will discuss the achievement of the research and talk about possible future work that can be done to further improve the approach.

6.2 Review of the goal

As we discussed earlier, the most important aspect of safety-critical interactive systems is correctness. When a safety-critical interactive system is developed, there is a communication gap between the business stakeholders who give out the requirements, and the developers who construct the formal specifications as a guide of the development process. We can't guarantee that the formal models meet the defined requirements. Thus the system might not be what the stakeholder expected, which is likely to cause new errors. Our goal was to

find a way to support the formal models construction by using the behaviour specifications given by the stakeholders.

6.3 Contribution

The contribution made in this thesis is an approach that transforms behaviour specifying Cucumber Scenarios, into First-Order-Logic predicates that describe the scenarios. These First-Order-Logic predicates can be used to define the requirements of the formal models, or used with modification as attributes of the models. This approach is different from the typical use of creating tests based on the Cucumber Scenarios, it supports the development of the formal Z models.

6.4 Limitations

There are some limitations of this approach, the first one is that the converter could only handle one to one actions. If there are multiple targets, the converter can't recognise which is the subject or which is the object. The second limitation is that the action predicates from the scenarios must be checked manually, to be turned into functional predicates in the Z specifications. The third limitation is that the transformed predicates can't be automatically model checked, which means we can't formally prove that the output First-Order Logic predicates belong to the actual Z models. The fourth limitation is that the converter can't differentiate "belongs to" and "is", for example:

```
Given lollipop is candy
```

In our converter, this will be translated into "lollipop = candy". This predicate is not completely true in formal specifications, lollipop is a subset of candy, but they are not equal. This result is because the Cucumber Scenarios are written in natural language, whose meaning heavily relies on the grammar and context of the writer. The fifth limitation is the approach only considered simple predicates such as equality, action (such as add(a,b)) and exists(∃), the converter didn't consider complex relationships or declarations such as every (∀).

Finally, the approach suffers from the general limitations of model checking techniques, the size of the predicates depends on the complexity of the Cucumber Scenarios, and the approach needs a fairly big amount of data to cover the different aspects of the system under development.

6.5 Future work

The approach can be improved by considering more symbols and quantifiers, both in Z specifications and First-Order-Logic. By doing this, we can extend the meaning of the predicates to suit more complex scenarios and requirements. In additional, the approach can be improved by changing the transformation algorithm, so the steps (such as Sky is blue) are transformed into First-Order Logic in a smarter way.

Furthermore, this project used grammar rules created by ourselves, which may not be used by the other users, in the future we can modify the transformer to use grammar that is more popular, such as Combinatory Categorial Grammar (CCG) [26], this will help us to make the approach more applicable for other users.

Finally, the output predicates can be improved so that the tool can automatically prove, that the output predicates belong to the formal specifications of

the system. A automatic model checking process can also be implemented for the output predicates to make sure that they are correct.

6.6 Conclusion

In this research, we investigated the related work of Formal Methods including Model-Driven-Development and Model-Based Testing as well as Z and Alloy. We also explored Behaviour Driven Development techniques including Cucumber framework. We conducted research and implemented a converter that converts Cucumber Scenarios into First-Order-Logic predicates, by modifying the scenarios according to the rules we set up and manipulating the step texts. Despite the limitation we previously discussed, the converter can output useful First-Order-Logic predicates, that can help with supporting the development of formal Z models.

Bibliography

[1] Acceptance test driven development (atdd). https://www.agilealliance.org/glossary/atdd/, Last accessed 17 March 2019.

[2] Alloy in prob. https://www3.hhu.de/stups/prob/index.php/Alloy, Last accessed 20 June 2019.

[3] Business process model and notation. https://www.omg.org/spec/BPMN/2.0/, Last accessed 22 March 2019.

[4] Cucumber - scenario outline. https://www.tutorialspoint.com/cucumber/cucumber_scenario_outline.htm, Last accessed 18 June 2019.

[5] Cucumber and scenario outline. https://www.baeldung.com/cucumber-scenario-outline, Last accessed 18 June 2019.

[6] File system lesson i. http://alloytools.org/tutorials/online/frame-FS-1.html, Last accessed 18 June 2019.

[7] Foundational uml (fuml) reference implementation. http://modeldriven.github.io/fUML-Reference-Implementation/, Last accessed 21 March 2019.

[8] Fuzz typechecker for z. https://spivey.oriel.ox.ac.uk/corner/Fuzz_typechecker_for_Z, Last accessed 21 March 2019.

[9] Graphviz - graph visualization software. http://www.graphviz.org/, Last accessed 20 June 2019.

[10] Writing scenarios with gherkin syntax. https://hiptest.com/docs/writing-scenarios-with-gherkin-syntax/, Last accessed 10 June 2019.

[11] What is domain-driven design, 2007. http://dddcommunity.org/learning-ddd/what_is_ddd/, Last accessed 17 March 2019.

[12] Ramadan Abdunabi, Wuliang Sun, and Indrakshi Ray. Enforcing spatio-temporal access control in mobile applications. *Computing*, 96(4):313–353, 2014.

[13] Jean-Raymond Abrial. The B tool (abstract). In Robin E. Bloomfield, Lynn S. Marshall, and Roger B. Jones, editors, *VDM '88, VDM - The Way Ahead, 2nd VDM-Europe Symposium, Dublin, Ireland, September 11-16, 1988, Proceedings*, volume 328 of *Lecture Notes in Computer Science*, pages 86–87. Springer, 1988.

[14] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the " small scope hypothesis ". 2002.

[15] Chitta Baral, Marcos Alvarez Gonzalez, and Aaron Gottesman. The inverse lambda calculus algorithm for typed first order logic lambda calculus and its application to translating english to FOL. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2012.

[16] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] Peter Bittner Benno Rice, Richard Jones and Jens Engel. Welcome to behave!, 2014. `https://behave.readthedocs.io/en/latest/`, Last accessed 18 March 2019.

[18] Judy Bowen and Swikrit Khanal. Test stub generation from interaction and behavioural models. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2018, Paris, France, June 19-22, 2018*, pages 7:1–7:6. ACM, 2018.

[19] Judy Bowen and Steve Reeves. Formal models for informal GUI designs. *Electr. Notes Theor. Comput. Sci.*, 183:57–72, 2007.

[20] Judy Bowen and Steve Reeves. Ui-design driven model-based testing. *ECEASST*, 22, 2009.

[21] Judy Bowen and Steve Reeves. Ui-driven test-first development of interactive systems. In Fabio Paternò, Kris Luyten, and Frank Maurer, editors, *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011, Pisa, Italy, June 13-16, 2011*, pages 165–174. ACM, 2011.

[22] Ricky W. Butler. What is formal methods? https://shemesh.larc.nasa.gov/fm/fm-what.html/, Last Updated: 10 April 2016, Last accessed 03 June 2019.

[23] Joe Carlson. Two additional patient deaths linked to medtronic infusion pump, NOVEMBER 1, 2016. http://www.startribune.com/2-additional-patient-deaths-linked-to-medtronic-infusion-pump/399576491/, Last accessed 03 June 2019.

[24] John Douglas Carter and William Bennett Gardner. Bhive: Behavior-driven development meets b-method. In Stuart H. Rubin and Thouraya Bouabana-Tebibel, editors, *Quality Software Through Reuse and Integration*, pages 232–255, Cham, 2018. Springer International Publishing.

[25] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279. ACM, 2000.

[26] Stephen Clark and James R. Curran. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552, 2007.

[27] Christian Colombo, Mark Micallef, and Mark Scerri. Verifying web applications: From business level specifications to automated model-based testing. In Holger Schlingloff and Alexander K. Petrenko, editors, *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.*, volume 141 of *EPTCS*, pages 14–28, 2014.

[28] Maximiliano Cristiá and Pablo Rodríguez Monetti. Implementing and applying the stocks-carrington framework for model-based testing. In Karin K. Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 167–185. Springer, 2009.

[29] Emina Torlak Eunsuk Kang Joe Near Daniel Jackson, Aleksandar Milicevic. Alloy. `http://alloytools.org/`, Last accessed 22 April 2019.

[30] Antoni Diller. *Z - an introduction to formal methods*. Wiley, 1990.

[31] Martin Fowler. Domain specific language, 2008. `https://martinfowler.com/bliki/DomainSpecificLanguage.html`, Last accessed 23 March 2019.

[32] Theodore Hellmann. Leet (leet enhances exploratory testing). ,`https://archive.codeplex.com/?p=leet`, Last accessed 03 April 2019.

[33] Theodore D. Hellmann, Ali Hosseini-Khayat, and Frank Maurer. Supporting test-driven development of graphical user interfaces using agile interaction design. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 444–447. IEEE Computer Society, 2010.

[34] Theodore D. Hellmann, Ali Hosseini-Khayat, and Frank Maurer. Test-driven development of graphical user interfaces: A pilot evaluation.

In Alberto Sillitti, Orit Hazzan, Emily Bache, and Xavier Albaladejo, editors, *Agile Processes in Software Engineering and Extreme Programming - 12th International Conference, XP 2011, Madrid, Spain, May 10-13, 2011. Proceedings*, volume 77 of *Lecture Notes in Business Information Processing*, pages 223–237. Springer, 2011.

[35] ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics.* Prentice-Hall International series in computer science. ISO/IEC, first edition, 2002.

[36] Daniel Jackson. Micromodels of software: lightweight modelling and analysis with alloy. 01 2002.

[37] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012.

[38] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 730–733. ACM, 2000.

[39] Selina Gyde Judy Bowen. Pimed. An editor for presentation models and presentation interaction models,`https://sourceforge.net/projects/pims1/`, Last accessed 05 April 2019.

[40] Cem Kaner. A tutorial in exploratory testing. *Tutorial presented at QUEST2008.(Available online at: http://www. kaner. com/pdfs/QAIExploring. pdf, accessed: 26 Jan 2014)*, 2008.

[41] DANIELLE KIRSH. Focusing on usability can limit medical device recalls: Here's how, JANUARY 17, 2019. https://www.massdevice.com/focusing-on-usability-can-limit-medical-device-recalls-heres-how/, Last accessed 03 June 2019.

[42] Leslie Lamport. *LaTeX - A Document Preparation System: User's Guide and Reference Manual, Second Edition*. Pearson / Prentice Hall, 1994.

[43] Ioan Lazar, Simona Motogna, and Bazil Pârv. Behaviour-driven development of foundational UML components. *Electr. Notes Theor. Comput. Sci.*, 264(1):91–105, 2010.

[44] Michael Leuschel and Michael J. Butler. Prob: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003.

[45] Nan Li, Anthony Escalona, and Tariq Kamal. Skyfire: Model-based testing with cucumber. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, pages 393–400. IEEE Computer Society, 2016.

[46] Cucumber Limited. Cucumber. https://docs.cucumber.io/gherkin/, Last accessed 18 March 2019.

[47] D. Lubke and T. van Lessen. Modeling test cases in bpmn for behavior-driven development. *IEEE Software*, 33(05):15–21, sep 2016.

[48] Petra Malik, Lindsay Groves, and Clare Lenihan. Translating Z to alloy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2010.

[49] Karleigh Moore and Dishant Gupta. Finite state machines. `https://brilliant.org/wiki/finite-state-machines/`, Last accessed 20 March 2019.

[50] Dan North. Introducing bdd, 2006. `https://dannorth.net/introducing-bdd/`, Last accessed 18 March 2019.

[51] Dan North. How to sell bdd to the business, 2009. `https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business#showModal?modal-signup-complete/`, Last accessed 18 March 2019.

[52] Daniel Plagge and Michael Leuschel. Validating Z specifications using the probanimator and model checker. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer, 2007.

[53] Tahina Ramananandro. *Mondex* , an electronic purse: specification and refinement checks with the *Alloy* model-finding method. *Formal Asp. Comput.*, 20(1):21–39, 2008.

[54] S. Rose, M. Wynne, and A. Hellesoy. *The Cucumber for Java Book: Behaviour-driven Development for Testers and Developers*. Pragmatic programmers. Pragmatic Bookshelf, 2015.

[55] Colin F. Snook, Thai Son Hoang, Dana Dghaym, Michael J. Butler, Tomas Fischer, Rupert Schlick, and Keming Wang. Behaviour-driven formal model development. In Jing Sun and Meng Sun, editors, *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings*, volume 11232 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2018.

[56] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[57] Chris Stevenson. Testdox. `http://agiledox.sourceforge.net/index.html`, Last accessed 17 April 2019.

[58] Phil Stocks and David A. Carrington. A framework for specification-based testing. *IEEE Trans. Software Eng.*, 22(11):777–793, 1996.

[59] Emmanuel M. Tadjouddine and Frank Guerin. Verifying dominant strategy equilibria in auctions. In Hans-Dieter Burkhard, Gabriela Lindemann, Rineke Verbrugge, and László Zsolt Varga, editors, *Multi-Agent Systems and Applications V, 5th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2007, Leipzig, Germany, September 25-27, 2007, Proceedings*, volume 4696 of *Lecture Notes in Computer Science*, pages 288–297. Springer, 2007.

[60] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.

[61] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.

[62] Jack C. Wileden, John H. Sayler, William E. Riddle, Alan R. Segal, and Allan M. Stavely. Behavior specification in a software design system. *Journal of Systems and Software*, 3(2):123–135, 1983.

Appendices

Appendix A

Cucumber Scenarios

Here is the Cucumber Scenarios that describe the expected behaviour of the banking system in the form of "Given", "When", "Then" and the optional "And".

```
Feature: Operation on the account
Banking operations such as depositing (adding) or
withdrawing (taking) money from an account.
Or switching account types such as from new to cheque
#Should do relevant effect to the account
#when operations today don't exceed 5

#Attempt to deposit money when operations today don't exceed 5
Scenario: Successfully deposit some money to an account
#Less than 5
Given Operationtoday is 4
And The current bank credit is 15
When The amount is 5
Then add amount to credit
```

```
And add 1 to Operationtoday
#The credit should change
Then The credit should be 20
And Operationtoday should be 5

#Attempt to deposit money when operations today exceed 5
Scenario: Unsuccessfully deposit some money
to an account with too high operationsToday
Given Operationtoday is 5
And  The current bank credit is 15
When The amount is 4
Then The credit is not changed
Then The credit should be 15
And Operationtoday should be 5

# Attempt to withdraw money when operations today don't exceed 5
Scenario: Successfully withdraw some money from account
#Less than 5
Given Operationtoday is 3
And  The current bank credit is 12
When The amount is 4
Then subtract amount from credit
And add 1 to Operationtoday
#The credit should change
And The credit should be 8
And Operationtoday should be 4

#Attempt to withdraw money when credit < amount
Scenario: Unsuccessfully withdraw some money
```

```
from account with insufficient credit
Given Operationtoday is 4
And  The current bank credit is 4
When The amount is 5
Then The credit is not changed
#The credit should remain the same
Then The credit should be 4
And Operationtoday should be 4


#Attempt to switch the account type from new to
#cheque when operations today don't exceed 5
Scenario: Successfully switch accounttype
from new to cheque
Given Accounttype is new
And Operationtoday is 0
Then Paul does switch Accounttype to cheque
And add 1 to Operationtoday
Then Accounttype should be cheque
And Operationtoday should be 1

#Attempt to switch the account type from cheque
#to saving when operations today don't exceed 5
Scenario: Successfully Switch account type
from cheque to saving
Given Accounttype is cheque
And Operationtoday is 4
Then Nancy does switch Accounttype to saving
And add 1 to Operationtoday
```

```
Then Accounttype should be saving
And Operationtoday should be 5


#Attempt switch account type because operations today exceed 5
Scenario: Switch account type from saving to cheque
Given Accounttype is saving
And Operationtoday is 5
#These part shouldn't have any effect
Then Bowen does switch Accounttype to saving
And add 1 to Operationtoday
#Account type should remain the same
Then Accounttype should be saving
And Operationtoday should be 5
```

Appendix B

Z Specification

Here is the Z specification of the banking system, note that it is a separate Z version of the banking system instead a translation based on the Cucumber BDD scenarios.

$$TYPE ::= Cheque | Saving | New$$

$$\begin{array}{|l} OperationLimit : \mathbb{N} \\ AmountLimit : \mathbb{N} \\ CreditLimit : \mathbb{N} \\ \hline OperationLimit = 6 \\ AmountLimit = 6 \\ CreditLimit = 21 \end{array}$$

BankAccount

$Credit : \mathbb{N}$

$OperationToday : \mathbb{N}$

$AccountType : TYPE$

$Credit > 0$

$Credit < CreditLimit$

$OperationToday \geq 0$

$OperationToday < OperationLimit$

BInit

$BankAccount$

$Credit = 2$

$OperationToday = 0$

$AccountType = New$

Invariant

$Credit' = Credit + amount$

DepositMoney

$\Delta BankAccount$

$Amount? : \mathbb{N}$

$\neg(AccountType = New)$

$Amount? > 0$

$Amount? < AmountLimit$

$Credit' = Credit + Amount?$

$OperationToday' = OperationToday + 1$

$AccountType' = AccountType$

WithdrawMoney

$\Delta BankAccount$

$Amount? : \mathbb{N}$

$\neg(AccountType = New)$

$Amount? > 0$

$Amount? < AmountLimit$

$Credit' = Credit - Amount?$

$Credit > Amount?$

$OperationToday' = OperationToday + 1$

$AccountType' = AccountType$

SwitchToSaving

$\Delta BankAccount$

$Credit' = Credit$

$(OperationToday' = OperationToday + 1)$

$\neg(AccountType = Saving)$

$AccountType' = Saving$

SwitchToCheque

$\Delta BankAccount$

$Credit' = Credit$

$(OperationToday' = OperationToday + 1)$

$\neg(AccountType = Cheque)$

$AccountType' = Cheque$

DayPass

$\Delta BankAccount$

$\neg(AccountType = New)$

$Credit' = Credit$

$OperationToday' = 0$

$AccountType' = AccountType$

GainInterest

$\Delta BankAccount$

$(Credit' = Credit * 2) \land AccountType = Saving$

$OperationToday' = OperationToday + 1$

$AccountType' = AccountType$