Integrating Behavioural and Formal Specifications

A thesis

submitted in fulfilment

of the requirements for the degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

BOWEN LIU

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2024

Abstract

Behaviour-driven development and formal methods have been shown to be effective techniques for different parts of the interactive system development process. However, when the different parts of an interactive system are combined, issues may arise due to the different focuses of the techniques. For a safety-critical interactive system, where safety is particularly emphasised, using either technique individually does not cover all aspects of the system. Thus a complementary approach of integrating behaviour-driven development and formal methods, would be better suited for developing safety-critical interactive systems.

This thesis presents an investigation into a method that integrates formal methods and behavioural specifications. This method enables the identification of any inconsistencies between the behavioural specification and the Z specification, allowing potential problems to be found and fixed earlier. We also demonstrate how our method supports ensuring consistency after refinement of the Z specification.

Acknowledgements

My sincere thanks to my chief supervisor Judy Bowen for her continued support, patience and enthusiasm. I could not have done this work without her guidance. I would also like to thank my mother for her mental and financial support, she has always been supportive even in my hardest times. Thanks also to my co-supervisors Jessica Turner and Steve Reeves, for their input, feedback and advice throughout the process. Lastly, I am grateful to the University of Waikato and the University of Waikato Computer Science Department for providing financial support.

I acknowledge the use of Grammarly language tool at the final stage of preparing my thesis. I used Grammarly to check my grammar including punctuation, word choice and spelling.

Contents

1 Introduction **1**

1.0.1 Terminology . . . 3

1.1 Techniques for Developing Interactive Systems . . . 4

1.1.1 Human Centered Design . . . 5

1.1.2 Formal Methods . . . 8

1.2 Problem Identification . . . 10

1.3 Thesis Statement . . . 11

1.4 Motivation . . . 11

1.5 Thesis Structure . . . 14

2 Background **16**

2.1 Behaviour-Driven Development . . . 16

2.1.1 Cucumber . . . 19

2.1.2 Behavioural Specification . . . 19

2.1.3 Behaviour-Driven Development for Interactive Systems . 21

2.2 Formal Methods . . . 23

2.2.1 Formal Models . . . 24

2.2.2 Selection of Formal Notation . . . 25

2.2.3 Z Notation . . . 28

2.2.4 Formal Methods in Safety-critical Interactive Systems Development . . . 32

2.3 Combining Behaviour-driven Development and Formal Methods 36

2.3.1 Combining Formal and Informal Methods 36
2.3.2 Comparison Between Informal and Formal Methods . . . 38
2.4 Summary 43

3 Comparing Behavioural and Formal Specifications 44
3.1 Methodology 44
3.1.1 Consistency 50
3.2 Terminology 51
3.3 Comparing the Specifications 52
3.3.1 Step One Finding Essential Facts 53
3.3.2 Step Two Identifying Goal-Related Operations and Finding Matching Statements 58
3.3.3 Step Three Analysing Essential Facts and Forming Predicates 64
3.3.4 Step Four Checking Predicates 65
3.4 Summary 67

4 Infusion Pump Example 68
4.0.1 Behavioural Specification 69
4.0.2 Z Specification 72
4.1 Applying the Process 72
4.1.1 Step One Finding Essential Facts 72
4.1.2 Step Two Identifying Goal-Related Operations and Finding Matching Statements 78
4.1.3 Step Three Analysing Essential Facts and Forming Predicates 95
4.1.4 Step Four Checking Predicates 103
4.2 Summary 105

5 Discussion 107
5.1 Negative Scenarios 107

5.2 Sequence and Ordering of Events . . . 111
5.3 Refinement . . . 115
5.3.1 Data Refinement . . . 117
5.3.2 Operation Refinement . . . 120
5.3.3 Refinement in Our Process . . . 121
5.4 Summary . . . 121

6 Conclusions **122**
6.1 Overview . . . 122
6.2 Contributions . . . 124
6.3 Limitations and Future Work . . . 125
6.4 Conclusion . . . 128

Appendices **130**

A NiKi T34 Infusion Pump Z Specification **131**

Bibliography **156**

List of Figures

3.1 Comparative analysis step in development process 45

4.1 T34 infusion pump 69

Table of Listings

1 A task scenario for buying a digital camera 6

2 A behavioural specification example 7

3 Test-driven development 17

4 BDD scenario 18

5 Feature description 20

6 Scenario example 20

7 Schema example 29

8 State schema 29

9 Operation schema example 30

10 Account state schema 31

11 Deposit money operation example 31

12 Type definitions 31

13 Axiomatic definitions 32

14 Process of comparing behavioural and Z specifications 53

15 A vending machine scenario for buying a snack using money . . 54

16 Pattern for finding essential facts 55

17 Picking an operation 58

18 BuyVM operation 59

19 Pay and Take operations 59

20 Where to find matching statement 60

21 Generalised scenario Buying a snack using money 64

22 Feature for loading the syringe into the driver 70

23 Feature for setting up the infusion 71
24 Feature for locking the program 71
25 Feature for pausing the infusion 72
26 Goals of the features for the T34 73
27 Assumption list for T34 after Goal One 74
28 Assumption list for T34 after Goal Two 76
29 Assumption list for T34 after Goal Three 77
30 Assumption list for T34 after Goal Four 78
31 Preloading operation 79
32 T34 system schema 80
33 ReadyToInfuse operation 83
34 Step Two for set up the infusion - Part One 84
35 Step Two for set up the infusion - Part Two 86
36 Setting up the infusion with the parameters 87
37 Generalised setting up the infusion with the parameters 88
38 Generalised assumption list for set up the infusion 88
39 ProgramLock operation 89
40 Step Two for lock the program 90
41 PauseInfusion operation 91
42 Step Two for pause the infusion 91
43 Assumption list for load the syringe into the driver 96
44 Predicate list for load the syringe into the driver 97
45 Generalised assumption list for set up the infusion 97
46 Predicate list for set up the infusion 100
47 Predicate list for T34 infusion pump example 101
48 A vending machine scenario for buying a snack using money . . 108
49 Vending machine scenario with no cookie in stock 108
50 A banking system scenario for logging in using password 112
51 Dan North scenario draft 113

52 Dan North withdraw cash scenario 114
53 Data type YesNo 118
54 Data type Level 118
55 Retrieve relation from BatteryLevel to BatteryYesNo 119

Chapter 1

Introduction

This thesis presents an investigation into the integration of formal methods and behavioural specifications for the development of safety-critical interactive software systems. Interactive systems require the user and computer to exchange information, they allow interactions between systems and humans. Safety-critical interactive systems are a subset of interactive systems, their failure may result in unacceptable damage, such as death or serious injury to people. Examples of safety-critical interactive systems include banking applications, medical infusion pumps, and nuclear power plant control systems.

Interactive systems consist of different aspects including user, front end, and back end. The user is the person who interacts with the interactive system. The front end is where the user and the system exchange information, it is the combination of the visual elements (what the user sees), and the interactive elements (where the user interacts with the system), the front end is usually called the user interface. The back end is where the system processes the input information received from the user and generates the output response to the user.

A few attributes should be considered in interactive system development. As the systems must support the goals of the users, the users should be involved throughout the design and development process. Interactive systems should be functional and effective to use, as well as easily understandable for the target audience to work efficiently without requiring extensive technical knowledge of the system. The users should be able to evaluate the results of their interactions with the system easily. As the developers and users of the system are usually two separate groups of people, consistency should be preserved at all times to ensure that the system the developers are designing and the goals the users want to achieve do not contradict each other. In addition, the interactive system should be correct, which means there should not be failures or defects in processing the input and giving the output. However, we can not guarantee that the system will be completely correct. In this research, we aim to reduce errors, by identifying the inconsistency between the developed system and the goals the users want to achieve.

Many other attributes need to be considered when developing interactive systems, but in this thesis, we address user requirements, consistency, and correctness. Building any interactive software system is complex as the attributes discussed above need to be satisfied. In this work, we introduce some additional constraints by focusing on safety-critical interactive software systems. Systems that require a high level of correctness and security, such as medical devices or nuclear power plants, are called safety-critical systems [66].

Failures in safety-critical interactive systems can lead to serious damage to people, property, or the environment. For example, failures in a medical infusion pump can cause severe damage to patients or even kill them. Failures in a bank system can cause a loss in finances or even damage the economy. The

failure of safety-critical systems is often deemed unacceptable, thus developers want to ensure that safety-critical systems are correctly built and failures are unlikely to occur.

1.0.1 Terminology

From the software developer's perspective, the terms "errors, defects, bugs, failures, and faults" are often used interchangeably [68, 11], thus it is important to distinguish them before we discuss further, and we refer to definitions given in [68] which are:

- Failure is an unacceptable behaviour exhibited by a system. Any violation of requirements is considered a failure, such as incorrect output or a system running too slowly.
- Defect is a flaw in any aspect of the system including the requirements, the design, and the code, that contributes, or may potentially contribute, to the occurrence of one or more failures. A defect is also known as a fault.
- Error is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect into the system. An error can be made at any stage of the software development process, from requirements to implementation and maintenance.

"Bugs" usually refer to both "failures" and "defects", to avoid confusion, we will not use "bugs" in this work, so we can better distinguish "failures" and "defects".

As discussed, errors can lead to defects and further failures. In this work, we aim to reduce errors in safety-critical interactive systems development. Examples of errors in safety-critical interactive systems include misinterpretation of the requirements, and bad decision-making in the design phase. For example,

given an automated oxygen supply system in a spacecraft, the requirement states "the oxygen level in the cabin must be maintained between 19.5% and 23.5%". However, this requirement does not consider the scenario of a fire outbreak, and the oxygen level might need to be temporarily reduced under 19.5% to suppress the fire. The developed system may rigidly maintain the oxygen level within the specified range even under extreme circumstances, which could lead to a spreading fire and put the crew in danger.

In addition to errors that can be made by the designers and developers, there can be human mistakes when the system is operated by a user. Human mistakes are caused by a human whose interaction or interpretation was not intended, such as pressing the wrong button or misinterpreting a message. Human mistakes are a major factor in causing system failures [78], because of a mismatch between a human operator's comprehension of how the system works and how the system actually works [33]. Such mismatches can not be eliminated and thus human mistakes are inevitable, but human mistakes can be reduced by mitigating this mismatch. Therefore, a human operator is often an important factor to be considered in developing a safety-critical interactive system. Although we do not model human behaviour in the system analysis, it still needs to be considered in the design and development process.

1.1 Techniques for Developing Interactive Systems

There are various approaches for designing and developing interactive systems, including human-centered approaches, and model-driven approaches, these two types of approaches are fit for the requirements of safety-critical interactive systems, we will introduce these next.

1.1.1 Human Centered Design

One of the human-centered design approaches is User-Centered Design (UCD) [75] or User-Driven Development (UDD). User-Centered Design is a conceptual framework [65] that emphasises user-centered characteristics and values where the user is involved throughout the process of design and development. UCD focuses on understanding the needs and goals of users and making the product "usable". Rubin and Chisnell [81] describe usable as "the user can do what he or she wants to do the way he or she expects to be able to do it, without hindrance, hesitation, or questions." Rubin and Chisnell defined that, for a product or service to be usable, it should be: Useful, that is the product allows the user to achieve goals and is motivating to use; Efficient, the product allows the user to achieve goals quickly and accurately; Effective, the product allows the user to achieve goals correctly and easily; Learnable, the user can learn to use the product easily and quickly; Satisfying, the user's perceptions of the product is good, that using the product is satisfying to the user; Accessible, the target user should be able to use the product in any situation, even for users with disabilities or additional needs.

Participatory design and co-design are also concepts that focus on involving the users in the design process. Instead of designing for users, participatory design [83] promotes designing with users, it allows users to cooperate with designers and developers in the design and development process. Involving users more in the development compared to participatory design, co-design [84] requires users to participate in the creative idea-generation process, strengthening the connections between users and developers. One way of practising participatory design is to ask users to participate in usability test task scenarios, a task scenario is "a longer, narrative, description of the test task that motivates the

participant to complete it" [92]. The developer will give tasks to the users and set up a scenario, then the users will give their expectations of the system, explain how they will carry out the task, and give their ideas about the system [91]. There are many forms of test scenarios, an example of a task scenario [92] is shown in listing 1. Task scenarios describe a problem that the participant of the test needs to solve with existing resources or conditions, they help clarify the user needs and expectations.

Listing 1 A task scenario for buying a digital camera

Test task: Buy a digital camera
Task scenario: You're going scuba diving in Australia. You want to buy a camera that you can use to take pictures of fish around the coral reef. Your budget is £150 (maximum). You're open-minded about the brand but you've heard megapixels are important for image quality so you want as many of those as you can get. Use the Argos website (www.argos.co.uk) to find a suitable camera."

While participatory design and co-design involve users directly in the design and development process, the focus is ensuring user requirements are understood and met, not necessarily safety. The expected behaviours of the system are described freely by the users, which could lead to worse clarity, different users can have inconsistent ways of describing their expectations, which makes ensuring correctness harder.

Behavioural-driven development (BDD) is the UCD development methodology that we focus on in this work, it is based on describing how the system should behave when the user operates it. BDD is a development technique that implements the system by describing the behaviours of the system [76]. BDD is an Agile software development approach [37] that encourages fast and flexible communication between the developer and customer/end user. Stakeholders and users give behavioural specifications that describe how the system should behave under certain circumstances. An example of a behavioural specification

is shown in listing 2:

Listing 2 A behavioural specification example

```
Feature: Buying a product using coins
As a user
Bowen wants to buy a product using coins
So that Bowen can get a snack

Scenario: Buying a coke using a coin
Given vending machine is working
When Bowen inserts a coin
And Bowen selects coke from the machine
Then vending machine supplies coke
And Bowen gets a coke
```

The details of the structure used in behavioural specifications will be discussed in Chapter Two. This structure helps the business stakeholders to frame their requirements in a structured way while keeping the benefit of the use of natural language, that it is intuitive and does not require specialist knowledge, such as a programming language. The writer does not worry about how the inner program works (such as how the coin triggers the can movement), instead, the context, event(input) and outcome (output) are what the writer is interested in. Behavioural specifications do not describe how the system achieves a behaviour, instead, they describe how the system will behave under certain situations.

Behavioural specifications have a defined structure and keywords such as "given, when, and, then", but there is no defined syntax. This structure enables a more efficient way of expressing the user's needs. Behavioural specifications are easy to read and write, but due to the natural language used when writing behavioural specifications, they have a suggested meaning, which means readers can misinterpret them, thus we can not be certain we have correctly understood a behavioural specification. We will describe BDD in more detail later in Chapter Two.

Behavioural specifications are often used as a guide for developing (programming in particular) the program and testing the developed program. As behavioural specifications do not describe how the inner program of the system works, we need some other techniques that can support the development of the back-end functionality correctly and safely. We introduce formal methods next.

1.1.2 Formal Methods

Formal methods are a set of mathematically precise and rigorous techniques developed to reason about systems. Such techniques and systems that are rigorously specified, modelled, developed, analysed, and verified by the techniques are considered *formal*. Formal methods include any notation and language that has a defined syntax and semantics, and preferably also a reasoning logic. Formal methods can be used to construct abstract system models that unambiguously specify the functionality of the expected system. Ideally, the ambiguity in the specifications will be reduced after applying formal methods, which means the correctness of the developed system can be improved. Formal specifications are rigorously specified descriptions used in formal methods, to describe systems to be developed. Systems already implemented can also be specified using formal specifications for analysis. The defined syntax and semantics ensure that a formal specification has only one meaning. These specifications or models can be refined further into working systems. We will describe formal methods in more detail later in Chapter Two.

The back end of an interactive system takes input from the user interface and calculates any output to send back to the user. Formal specifications can be used to describe different states of the system using input and outputs,

thus formal methods are useful for developing the back end of the interactive system. For safety-critical interactive systems which have high demands for correctness, this is particularly useful.

However, using formal methods requires a high level of technical skill to construct, learn, and make use of formal models, thus formal methods are often considered costly in both time and resources [53]. The resulting models are typically not understood by business stakeholders as they are not intended to be used for this purpose. While formal methods are useful for planning and developing the back end of interactive software systems, formal methods do not help us necessarily with satisfying user requirements for the front end of the system. Users and the front end are usually not the focus of formal methods.

So far we have identified two parts of the interactive software system that need to be taken care of throughout the development, which are the front end and the back end. We have also introduced two possible software development methods formal methods and behavioural-driven development to develop the front end and back end of the system. It makes sense to use both formal methods and behavioural-driven development to develop a safety-critical interactive system. However, if we want to use formal methods and behavioural-driven development in conjunction, we need to consider how to ensure developers using the two techniques can communicate effectively with each other. Using both formal methods and behavioural-driven development is complicated, because the languages and notations used in formal methods and behavioural-driven development are very different. The developers of the front end using behavioural-driven development can have a very different focus to the developers of the back end using formal methods. Developers can not compare parts

of the developed system described using formal methods and BDD directly. This leads to the identification of the problem we seek to address.

1.2 Problem Identification

Formal methods and behavioural-driven development have been successfully deployed independently and shown to be effective for system development [17, 72]. However, due to the different focuses of formal methods and behavioural-driven development, using either technique individually does not cover all the aspects of a safety-critical interactive system. The demand for safety in safety-critical interactive systems requires more than either of the two techniques can offer alone.

Combining behavioural-driven development and formal methods will provide the benefit of involving users as well as higher level of correctness provided by formal methods. However, because the notation and languages of formal specifications and behavioural specifications are very different, we can not be certain that they describe the same system. Developers who have expertise in behavioural-driven development are less likely to be proficient at using formal methods, while formal method experts are less likely to be familiar with behavioural-driven development. The different notations along with the lack of defined semantics for BDD mean we cannot directly compare the two specifications and be certain they have the same meaning. This is the problem we seek to address in this work.

1.3 Thesis Statement

Based on the problems we have identified above and the desire to combine behaviour-driven development and formal methods, the following hypothesis is stated:

> **Hypothesis**
>
> Behavioural-driven development and formal methods can be used together to describe safety-critical interactive systems to support correctness. Behavioural specifications and formal specifications describing the same system can be compared to show that they are consistent and do not contradict each other. Behavioural specifications and formal specifications can be used together to give certainty that the proposed system will be consistent.

In order to prove this hypothesis, we consider the following research questions:

1. What are the challenges in comparing behavioural specifications with formal specifications?

2. How do we address the challenges identified in research question one to make behavioural specifications comparable with formal specifications?

3. How do we use the comparison in an integrated approach to support safety-critical interactive system development?

1.4 Motivation

When an interactive system is designed and developed, it is possible that the requirements described in the behavioural specification do not fully match the model described in the formal specification, which can lead to unwanted re-

sults. Suppose there is a conflict between the behavioural specification and the formal specification. The developers can get stuck when they combine the two specifications into the final system, because it is not clear which of the two specifications is correct.

A worse scenario is that the behavioural specification and the formal specification do not obviously conflict with each other when they are used together to build a system. However, errors may be introduced if there is a mismatch between the formal specification and the intended purpose or expectation of the behavioural specification, and such a mismatch will lead to undesirable results. Using a banking system as an example, the behavioural specification specifies: the bank system should automatically log out after five minutes of inactivity. A formal specification of this banking system specifies: the bank system automatically logs out after five minutes of inactivity. The two specifications are identical, both seem correct and not conflicted. However, there is a potential mismatch - the formal specification specifies that the system will count for five minutes, and after that, the system will log out; the behavioural specification states the same, but from a user perspective, the timer should reset when the user interacts within the 5 minutes interval. This mismatch could lead to an unexpected log out of the system.

Another possible mismatch is that the timer resets when a user interacts with the system within five minutes. However, an operation such as depositing a large amount of money may take more than five minutes to process. The user is not interacting with the system during the operation, but the system still logs out after 5 minutes of inactivity. Such behaviour of the system is not intended and could lead to financial loss. The different developers who wrote the behavioural specification and the formal specification will not find errors,

because the specifications they wrote have been reviewed and are technically correct when the specifications are considered separately. But since there is a different idea and focus about how something should behave between behavioural specification and formal specification, the resulting system could have unexpected behaviours.

Each developer group has the belief that they understand the whole system, but what they can be sure about is only their part of the system, depending on whether they are developing the front end or back end. The developers of the whole safety-critical interactive system have the belief that they understand how the whole system should behave, but when the separate parts are combined, things may go wrong.

The development of safety-critical interactive systems, despite significant resource investment, can result in suboptimal outcomes. Examples include the famous London Ambulance Service Computer Aided Despatch System (LASCAD) project failure [22]. The LASCAD project was responsible for receiving emergency calls for the ambulance services, despatching ambulances based on an understanding of the calls and the availability of resources, and monitoring the progress of the response to the calls [47]. The 1992 LASCAD consists of a few parts including front end and back end aspects. There were many problems with LASCAD - one was the lack of involvement of end users which meant front-end developers did not even necessarily understand the requirements, and the developers of the system did not realise the fact that the success of the system was dependent on an ideal 100% accuracy and reliability of the technology, as well as absolute cooperation from all people within this system. While the system appears well-designed on paper, achieving flawless execution in practice is often unrealistic.

A more recent example is the Post Office Horizon scandal in the UK [4]. Horizon was developed to manage transactions at Post Office branches across the UK, Horizon was required for calculating profits and losses at each branch. Between 1999 and 2015, more than 900 users were wrongly prosecuted for theft, fraud, and false accounting based on faulty information given by Horizon. One of the issues Horizon had was it freezes the screen when an operator tries to input money, so every time someone clicks on it subsequently, it repeats the transaction. Although overwhelming evidence was presented, the investigators of Post Office believed their systems could not be wrong [5].

The examples discussed show that even big projects with a high amount of resources spent can lead to failures without the involvement and input of end-users, structured development, and testing methods. Developers across all parts of a system as well as users must cooperate for the successful development and deployment of a safety-critical interactive system. The use of both behavioural-driven development and formal methods will help with such cooperation.

1.5 Thesis Structure

In Chapter Two, we start by discussing background knowledge of behavioural-driven development and formal methods in more detail, particularly focusing on their application in safety-critical interactive system development. We explain prior research that integrates informal approaches, in particular behavioural-driven development, with formal methods, and discuss how our research is relevant in bridging the gap between informal and formal specifications for safety-critical interactive systems.

In Chapter Three, we outline the proposed process for comparing behavioural and Z specifications, and checking consistency between them. We give a detailed step-by-step explanation of the process and explain how conditions described in the behavioural specification are identified and formalised, for comparison with the Z specification. In Chapter Four, we present a case study using the Niki T34 infusion pump, to demonstrate the application of our process for checking the consistency between behavioural and Z specifications.

In Chapter Five, we expand on the findings from exploring the process of comparing the behavioural and Z specifications, and address the challenges that emerged from the exploration. Finally, in Chapter Six, we provide an overview of this thesis, and discuss what has been achieved. The chapter summarises the key findings and contributions of this research, then discusses the limitations of the work and outlines possible future work.

Chapter 2

Background

This research covers two areas of software development: behaviour-driven development and formal methods, with an emphasis on using them jointly. We briefly introduced behaviour-driven development and formal methods in chapter one. In this chapter, we will extend these topics by discussing the background knowledge of behaviour-driven development and formal methods. After that, we will discuss how we can use these two methods together, and review related works.

2.1 Behaviour-Driven Development

Behaviour-driven Development (BDD) is a software development technique that describes the behaviours of the system [76]. BDD is an Agile software development approach [37] that encourages fast and flexible communication between the developer and customer/user. Agile is a set of frameworks and practices based on the values and principles expressed in the Manifesto for Agile Software Development [20]. Agile allows quick response to changes in requirements and adapts the solution to the changes by having business stakeholders

and developers work together daily. Agile involves the users and clients in the development, and promotes fast reaction to changes [20], which means the developers following an Agile approach adapt to new functions and demands well. However, Agile has its limitations: Agile methodology promotes changes in requirements [21], even in the late development phase. These changes could cause unexpected errors in the system as they are implemented, this limitation is vital to take into consideration in safety-critical interactive system development.

According to Dan north [76], BDD is inspired by Test-driven-development (TDD) [19], a software development process where software requirements are converted to test cases before coding. The process of TDD is shown in listing 3.

Listing 3 Test-driven development

1. Write a test that should fail
2. Code is modified to make the test pass
3. Add new tests and refactor the code

TDD is a simple but effective approach but it suffers from ambiguity - TDD relies on the testers to infer and write correct tests from the given requirements. Misunderstanding the requirements can lead to tests that do not fully capture the intended system behaviour, which potentially leads to faulty systems. To address the information gap between the requirements and the test writers, Dan North started using the word "behaviour" in place of "test", which led to the name "behaviour-driven development". BDD uses "behaviours" to represent requirements, in the form of scenarios, which provides background knowledge (conditions) for the behaviour, how the behaviour is performed, and what the result is. As BDD allows the stakeholders to give requirements using

the language of their own business domain, BDD makes it easier to communicate, between the technical domain user (software developers) and the business domain user (stakeholders).

Aiming to emphasise behaviour over testing, Dan North developed JBehave [39]. JBehave can 'run' behaviours as JUnit does with its tests. Stories of behaviours (scenarios) are configured into tests by mapping each step in the scenarios to Java methods, then these tests are executed and the results of the tests (whether they pass or not) are reported. In JBehave, all references to testing are removed and replaced by a vocabulary with references around 'behaviours'. This vocabulary of behaviours inspired the definition of a ubiquitous language for analysis, it is a structure used for BDD and is shown in listing 4.

Listing 4 BDD scenario

```
Title (one line describing the story)
Narrative:
As a [role]
 I want [feature]
So that [benefit]
#Acceptance Criteria: (presented as Scenarios)
Scenario 1: Title
Given [context]
  And [some more context]...
When  [event]
Then  [outcome]
  And [another outcome]...
```

This structure helps the business stakeholders to frame their requirements in a structured way while keeping the benefit of the use of natural language. Behavioural specifications do not describe how the system achieves this behaviour, instead, they describe how the system will behave under certain situations.

2.1.1 Cucumber

The BDD methodology and the acceptance criteria behaviours are later extended by the Cucumber framework [13], a popular tool which uses the Gherkin syntax [6]. Gherkin syntax is based on behavioural specifications [76] and is further extended to be reusable by using scenario outlines - a scenario template defined by the user with placeholders, so a scenario can be tested with different inputs for the placeholders. To use Cucumber, the tester writes tests and sets up conditions for each of the behavioural specification scenario steps to pass. Cucumber reads executable specifications written in natural language and then validates that the software does what those specifications say.

Cucumber automatically generates executable test stubs, but it only gives the structure of the tests, the test content still needs to be written manually. Gherkin is used in practice and research related to BDD. In [63], BDD scenarios using Gherkin syntax are translated into timed automata (mathematical models with timing constraints), to check compatibility between two artefacts. In [73], the set of steps of a scenario are shared within the feature to enable re-usability, to increase test effectiveness and efficiency. These works show that behavioural specifications can be used in testing methods and model checking to support better correctness. We discuss model checking further later in this chapter. As we will not be using the outline and reusability offered by cucumber scenarios in our work, we will use the behavioural specifications structure described in listing 4, with sections such as narratives, features, and scenarios, given by Dan North [76].

2.1.2 Behavioural Specification

The behavioural specification describes how a system behaves under certain circumstances. It can be used to define requirements from the perspective of

the users. A behavioural specification consists of a set of features related to the system to be developed, which contains some scenarios for each feature. Each feature in the specification describes a use case and states how the system should behave under the conditions described. A feature has feature descriptions as shown in listing 5.

Listing 5 Feature description

```
Feature title: a short text that describes the feature.
Narrative:
As a  <type of user>
I want <some goal>
So that <some reason>
```

The feature title should always describe an expected behaviour of the system, it should include the activity that a user wants to carry out and be easy to understand. For each feature, there should be a narrative describing the user, the goal, and the reason why the user wants to achieve the goal. Within each feature, multiple scenarios describe the outcomes, when an event triggers, with a given context. The context in a scenario should define all of, and no more than, the required context to achieve the outcome of that scenario. There may be both positive scenarios where the result is successful and negative scenarios where the result is unsuccessful. A scenario is shown in listing 6, the "And" clauses are optional. Note that the word "clause" means a group of words that contains a subject and a verb, such as "Given a can is opened", rather than a formal logic clause.

Listing 6 Scenario example

```
Scenario 1: Title (one line describing the behaviour)
Given [context]
And [some more context]...
When [event]
Then [outcome]
And [another outcome]...
```

The scenario title should describe what is different in this scenario from the

other scenarios, in other words, the scenario title should describe the "conditions" needed for the behaviour described in the scenario.

2.1.3 Behaviour-Driven Development for Interactive Systems

BDD has been used to support the understanding of system requirements and behaviours. Farooq et al. conducted a systematic review that shows that BDD helps to encourage the developers to focus on, and understand, the required behaviours of an application, by promoting the collaboration between the developers and business stakeholders [44]. Although requiring all the team members to collaborate and communicate the requirements correctly with each other is difficult, the use of behavioural specifications (scenarios) helps to specify the requirements. Behavioural specifications allow their writers to elaborate on their expectations of the interactive system, this is useful for the developers to understand what exactly the user demands, particularly for the front-end user interaction of the system.

One of the strengths of behavioural specifications is their structure. To leverage this structure, writers should carefully shape their way of writing, for a high-quality behavioural specification. Binamungu et al. conducted a survey on characterising the quality of behavioural specifications [23]. The practitioners surveyed support the quality principles proposed in the survey: Conservation of Steps (maximise the use of existing steps in the scenarios and avoid having too many steps only used in one or two scenarios); Conservation of Domain Vocabulary (use the vocabulary of a given business domain); Elimination of Technical Vocabulary (do not use technical terms only understandable to developers); Conservation of Proper Abstraction (the scenario should have an appropriate level of abstraction). The result of the survey shows that industry

teams from all around the world are currently using BDD to specify software requirements, with a common set of quality principles. In our work, we will adhere to these principles. The survey also points out that duplication of scenarios and steps can be introduced in large-scale BDD suites, by members joining the teams at different points in time, which could waste resources by doing duplicated work and even increase the chances of causing errors. We take note of this issue and avoid it by grouping scenarios, which will be discussed in Chapter Three.

Research has been conducted to explore how behavioural specifications can be used to support interface designs. Silva et al. presented a scenario-based approach to access user interface design artefacts in [88]. In their work, behavioural specification scenarios are compared with and tested against three artefacts - equivalent tasks modelled in HAMSTERS task models scenarios; equivalent interaction elements in Balsamiq [3] (a prototype mockup tool) user interface (UI) prototypes; and Balsamiq final UIs. HAMSTERS is a tool-supported graphical task modelling notation [71], and a task model is a structured representation used to describe activities that have to be carried out to achieve the user's goals [43]. Each of the steps in the behavioural scenarios corresponds to a task in the task model scenario, because the step and the task represent the same behaviour. The interaction elements in the UI prototype and the final UI, that will be affected by this behaviour, are accessed by the scenario-based approach. If the same behaviour is supported in both the UI prototype and the final UI, they are consistent. The corresponding interaction element in the UI prototype can be different from the corresponding interaction element in the final UI, as long as they support the same behaviour, otherwise they are inconsistent. The semantic analysis is supported by an ontology developed in [85], we discuss this ontology further later in section 2.3. Although limited to the HAMSTERS task models and Balsamiq

UI prototypes, the approach provides a way to evaluate the user interface using behavioural specifications and identify inconsistencies. Similarly, we use behavioural specifications to evaluate whether the expected behaviour of the system is achieved, and identify if there is any requirement in the behavioural specification not satisfied.

We have discussed the purpose of BDD and how BDD has evolved, we will discuss formal methods next.

2.2 Formal Methods

A formal method is defined by Bowen and Hinchey as: "a set of tools and notations (with formal semantics) used to specify unambiguously the requirements of a computer system that supports the proof of properties of that specification, and proofs of correctness of an eventual implementation, with respect to that specification" [29]. Although the use of formal methods aims to reduce the risk of errors in safety-critical system development, it is a common myth that using formal methods increases the cost of development. Hall addressed this myth [53] and explained that using formal methods decreases the cost of development, by helping the developers better understand the project being developed. Bowen and Hinchey support this opinion in [24] by showing a graph of investment in the requirements phase of NASA projects plotted against the cost of project overruns. This demonstrates that one of the benefits of using formal methods is to help get the requirements right, and hence reduce the cost of fixing errors later.

In [24], Bowen and Hinchey pointed out that, while formal methods are useful for software development, they should support rather than replace traditional development methods (structured design and development methods). Because

traditional methods also have their advantages, for example, BDD is easier to learn and can be used by business stakeholders without extensive knowledge and training. A major area for research as discussed in [30], is to find ways to integrate formal methods with other traditional development methods, our work lies within this area.

2.2.1 Formal Models

Formal models are rigorously specified mathematical abstractions, they are a technique used in formal methods to represent, specify and analyse (hence model) system behaviours. As suggested in [32], formal models provide an unambiguous way of reasoning about system properties, improving the following attributes: correctness (check that errors are not found), reliability (that proper service will continuously be delivered), and robustness (how well the system can cope with errors). Formal models can be mathematically proven to be correct, but a *correct* system is not necessarily a robust system.

Model checking [36] is an approach for automatically verifying a model of a system against its respective formal specification. Model checking can be used to find errors, but passing a check for a model does not mean the system is error-free. Inconsistencies between the model being checked and the formal specification of the system can be missed during the check, because the model is not guaranteed to capture every aspect of this system. Model checking provides a way to validate a model exhaustively, but the models can, in the end, be imperfect due to the potential mismatch between the user requirements and the final implementation, we can not guarantee the interpretation of the requirements is accurate. However, model checking does reveal the presence of errors when errors are reported by the check, thus it is useful for improving the correctness of the system. In our research, we evaluated a selection of formal notations for modelling interactive systems, we will discuss the evaluation next.

2.2.2 Selection of Formal Notation

As discussed, this research aims to increase the correctness of safety-critical interactive systems. We want to combine behavioural and formal specifications, so we can better understand the requirements and reveal errors in the process. For the formal notation we use in this research, we want to use a **declarative** notation that supports **free types**.

Declarative Notation

The goal of this research is to combine and compare behavioural and formal specifications, thus it is better to pick a declarative formal notation that resembles the characteristics of a behavioural specification: a descriptive language with a precondition (Given), an action (When), and a postcondition (Then), a scenario that describes how the system should behave under the circumstances.

Free Types

As a behavioural specification is written in natural language, data represented in the scenarios can be specified as basic types, such as "car". A basic type's internal structure is invisible and this means the data are not defined (for example, we do not know what possible value is within type "car"), the values with this basic type are not distinct [94] (there can be two "SUV" in the type "car"). A **free type** can make sure the values within this type are distinct, that each value is non-duplicate. Therefore, the formal notation we choose should support free types or a similar data structure.

It is worth noting that we only intend to specify systems without timing constraints (conditions that dictate the acceptable time interval). Support for real-time systems (systems designed to perform tasks and respond to events

within a specified, predictable time frame) is not considered.

Based on the requirements discussed above, we evaluated five formal notations for specifying, modelling, and verifying the safety-critical interactive systems. The options include B-method [16], Z notation [90], Vienna Development Method Specification Language [62, 38] (VDM-SL), Alloy [58] and PVS specification language [77, 82]. The result of the evaluation is shown in table 2.1. We describe each of these next.

	B	Z	VDM-SL	Alloy	PVS
Model Style	Imperative	Declarative	Declarative	Declarative	Declarative
Free Types	Yes	Yes	Yes	Yes	Yes
Tool Support	Yes	Yes	Yes	Yes	Yes

Table 2.1: Comparison of formal notations

B-method is "a means for specifying, designing and coding software systems" [16] created by Abrial. The B-method allows developers to start with a very abstract model and gradually add details to the first model. Each step can be formally verified. Specifications in B language usually have an imperative and operational style, they can provide a sequence of instructions to explain how to accomplish something. B-method does not support free types, but free types can be indirectly achieved by creating enumerated types using sets and constants. B-method is supported by model checkers Atelier B [2] and ProB [8].

Z notation [89] is a formal specification notation developed by Abrial in the late 1970s. Z notation models a system by using a number of sets to represent the system's data, then these sets of data can be changed to represent behaviours of the system. Z supports free types by defining a new type with distinct values. Specifications in Z usually have a declarative style as they describe what the system should achieve without specifying how the system

achieves that. Z is supported by model checkers including Z/EVES [14], ZTC [61] and ProZ (an extension of ProB) [9].

Vienna Development Method Specification Language [38] is a formal notation for Vienna Development Method [62], a formal method primarily designed for specifying and developing functional aspects of software systems [80]. Free types are supported in VDM-SL by the use of union types that contain all the elements from each of the components of the union type [38]. Specifications created using VDM-SL are usually declarative by focusing on what a system should do rather than how it should be implemented. VDM-SL is supported by model-checkers VDMTools [12] and Overture [7].

Alloy [59] is a descriptive specification language inspired by the Z specification language as discussed in [58]. The notation was created by Jackson et al. Alloy comes with an analyser called Alloy analyser [59]. Specifications created using Alloy notation models usually have a declarative style. Free types can be achieved in Alloy by the use of 'abstract' signatures that are basically any type, subtypes can be defined as extensions of this basic type. Different from the other notations, Alloy uses a finite scope check - the check is not exhaustive, a scope (size) must be specified for the model to be checked, the analysis never returns false positives but it only checks up to the given scope. However, the model is complete up to the set scope according to the editors' tutorial [1]. Although the limited checking scope exists in other notations and tools in the form of hardware/software capability, Alloy promotes the limited scope to get quick bug detection and simplified analysis.

PVS specification language [82] is a formal notation used to specify and analyse computer systems, based on simply typed higher-order logic [18]. Types

can be defined from base types such as booleans and numbers, including uninterpreted (with minimum assumption on the type) and interpreted (type expressions such as $[int- > int]$ [82]) type declarations, etc. PVS language comes with PVS tool, a mechanised environment for constructing and maintaining formal specifications and verifications. PVS is also supported by PVSio [74] for the simulation of user interactions, in the form of input/output operations. PVS language provides high expressiveness via the use of higher-order logic. However, the style of PVS specification is closer to a programming language than a behavioural specification.

The reason why these notations were considered is that they satisfy or partially satisfy the needs we discussed. As shown in table 2.1, Z, VDM-SL, Alloy and PVS all support a declarative modelling style and free types. As Alloy does not provide exhaustive model checking, it does not suit our needs for develop a safety-critical system. PVS language is closer to a programming language in the sense of how a system should be implemented, this is not suitable for an easier comparison with behavioural specification, which describes how a system should behave. VDM-SL's research community is not as active as Z's research community. Thus after careful consideration, Z notation is selected to specify the requirements of the safety-critical interactive system. We will discuss the Z notation in more detail next.

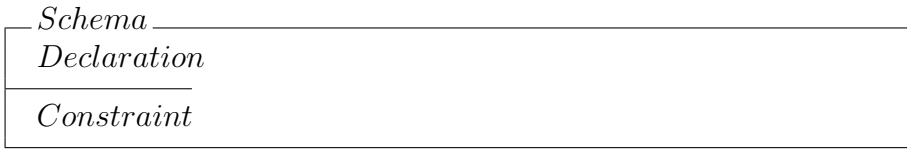
2.2.3 Z Notation

The Z notation [89] is a formal specification notation based on first-order predicate logic and typed set theory, that has been in use since the early 1980s [27]. Z notation can be used to describe the states of a system and operations that change those states. A state is "a representation of the possible values that a system might have" [28]. A state in a Z specification is modelled as a number of sets describing the data of the system. The system being modelled will

have an initial state which describes the data when the system is initialised. The operations describe the functions of the target system as changes in state. Each operation includes a "before" state and an "after" state, explaining how the data of the system changes when the operation pre-conditions are met.

A Z specification describes *what* a system is intended to do, but not *how* the system carries out the intended function. A Z specification describes an operation through its given pre-conditions and post-conditions. Z uses schemas to structure and compose descriptions of states and operations. A schema takes the form shown in listing 7.

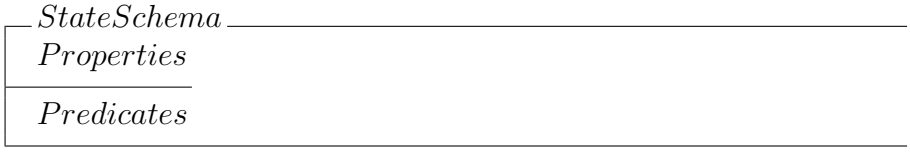
Listing 7 Schema example



Declarations describe compound objects or properties, constraints describe how the objects/properties are constrained. In the specifications used in our work, we mainly consider two types of schemas: state schema and operation schema.

There should be one state schema that represents the state space of the systems, as shown in listing 8.

Listing 8 State schema



A state schema with properties and predicates describes all the observations of state for the system, and the constraints on those properties.

There can be multiple operation schema, as the example shown in listing 9.

Listing 9 Operation schema example

$$
\begin{array}{|l}
\hline
OperationSchema \\
\Delta StateSchema \\
Input? : InputType \\
Output? : OutputType \\
\hline
Predicates \\
\hline
\end{array}
$$

An operation schema describes some effects upon the values of the state variables of a given state schema, it may have inputs and outputs (optional). The "Δ" (delta) symbol indicates StateSchema is changed by the operation, if we use a "Ξ" (Xi) symbol (Ξ StateSchema) instead, it means StateSchema is not changed by the operation. The predicates consist of two parts: the "before" state and "after" state. If the "before" state (pre-conditions) of the operation are met (if they are true), the system will change to the "after" state. The respective properties will change according to the post-conditions of the operation. The "after" state is indicated by the addition of "′" symbol to the observation name.

To demonstrate this further, consider the following example describing a bank account as shown in listing 10. The state schema called "Account" has a single property, "Money" that is a natural number, which represents the money value in the account.

Listing 10 Account state schema

$$
\begin{array}{|l}
\hline
Account \\
Money : \mathbb{N} \\
\hline
\end{array}
$$

An operation schema "DepositMoney" shown in listing 11 specifies adding some money to the account, if there is currently no money in the account.

Listing 11 Deposit money operation example

$$
\begin{array}{|l}
\hline
DepositMoney \\
\Delta Account \\
Input? : MoneyToBeDeposited \\
\hline
Money = 0 \\
Money' = Money + MoneyToBeDeposited \\
\hline
\end{array}
$$

If the current money value is 0, the amount of MoneyToBeDeposited will be added to the account.

Type is an important concept in Z, a type is a maximal set, so that each value "x" in a Z specification is associated with one and only one type: the largest set T where $x \in T$ [94]. The Z notation only has one built-in type $\mathbb{Z}$, that is the set of all integers. All other types will be constructed from $\mathbb{Z}$, or from basic types (a type whose internal structure is invisible) of values [94]. Z supports free types, which is a way to define a new type that can take on distinct forms. A Z specification may have definitions for free types (12) and axioms (13).

Listing 12 Type definitions

$$Type ::= PossibleValue1 | PossibleValue2$$

A type definition is a free type that describes the possible distinct values of an

observation with this type, this is important, as it ensures that PossibleValue1 is different from PossibleValue2, which eliminates the possibility of confusing these two values.

Listing 13 Axiomatic definitions

$$ObservationA : \mathbb{P}\,\mathbb{N}$$

$$ObservationA = 0 \,..\, 2$$

An Axiomatic definition is a free type with constraints on its possible values.

Z notation has an object-oriented extension that we considered called Object-Z introduced by Carrington et al. [34]. Object-Z enhances standard Z with some key features from object-oriented programming, such as classes, objects, and inheritance. By incorporating these features, Object-Z retains the advantages of standard Z and enables the specifications to be organised in a way that is closer to a real-world application, making the specification more modular and scalable. However, additional features comes with additional complexity; standard Z is already sufficient for the required criteria, using object-Z may introduce unnecessary difficulty to the comparison. Therefore, we decided that we will use standard Z notation for this research.

Next, we will discuss research on formal methods in safety-critical interactive system development, in particular Z notation.

2.2.4 Formal Methods in Safety-critical Interactive Systems Development

High-confidence means a high level of certainty and reliability associated with a system's performance, security, and accuracy. Integrated development envi-

ronments (IDEs) based on formal methods have been developed, to support the design and analysis of high-confidence user interfaces, this is particularly related to safety-critical system development. Fayollas et al. performed a comparison and evaluation of two state-of-the-art formal verification toolkits [46]: CIRCUS [45], a model-based development and analysis tool based on Petri net extensions, and PVSio-web [10], a prototyping toolkit based on the PVS theorem proving system. The result of the evaluation shows that, while both tools are useful for formal analysis, their analysis styles are complementary rather than competitive. Both tools contribute to aligning the interpretation of requirements with the models, but do not ensure consistency between the interpretation and models.

CIRCUS includes three tools: HAMSTERS, PetShop and SWAN. HAMSTERS [71] is used for editing and simulating task models. PetShop (Petri Net workshop) is used for developing systems using ICO (Interactive Cooperative Objects, a notation that uses Petri Nets [79], for describing behaviours of interactive systems. Petri nets are a graphical for formally describing systems, they can be used to model systems exhibiting concurrency in their operation [40]) notation. SWAN (Synergistic Workshop for Articulating Notations) is a tool for co-executing PetShop models and HAMSTER models by establishing correspondences between behaviours and tasks.

PVSio-web is a toolkit based on PVS. PVSio-web extends PVS by providing a graphical environment for formal methods of user interface, this is particularly useful for safety-critical interactive system development as PVSio-web allows developers to create device prototypes based on formal models. PVSio-web translates user actions over input widgets such as button presses into PVS expressions, which can be checked in PVSio [74] (a tool that extends PVS by

providing simulation of input and output).

In [46], a case study based on a subsystem of the Flight Control Unit (FCU) of the Airbus A380 is conducted for CIRCUS and PVSio-web. Each of these two tools is used to develop a prototype UI to capture the functionalities of the FCU software. In the prototype built in CIRCUS, a correspondence editing panel is used to establish the matching between the task model and the formal system model. The co-execution part of the SWAN tool provides support for validation to find inconsistencies between the two models such as some sequences of user actions are allowed by the system model but forbidden by the task model.

The PVSio-web analysis ensures internal consistency of the model by discharging proof obligations (a theorem stating that a property of a formal specification must hold true so that this specification is internally consistent), but the PVS theorems need to be defined that capture the requirements and properties. The accuracy is accessed by asking real Airbus cockpit experts to interact with the prototype and see if the prototype behaviour resembles that of the real system.

Interestingly, for the two examples developed in CIRCUS and PVSio-web in [46], there is no assurance of consistency between the interpretation of the requirements and the model (usability properties). They can check if the prototype and the defined interactions are consistent, but still do not have the ability to consider if the front-end and the back-end of the interactive system, as well as the artefacts that are used to describe these two parts of the system, are consistent.

Apart from the work discussed above, there has also been research done on using Z notation to support interactive system development. Duke et al. [42] conducted a case study on how Z notation can be used to express requirements on the interfaces of a media space (audio-visual communications environment), enhancing the user interface design with the mathematical rigor from Z. This case study suggests that Z notation can assist the development of user interfaces, which clarifies the requirements of user actions and the actions' effects.

Another work on formally describing the design of a user-interface was presented by Hussey in [56]. Hussey integrated Object-Z and User Action Notation (UAN) in this work, Object-Z is used for initial specification, then UAN is used to describe the interface and user interactions by directly annotating the existing specification. UAN is a user and task-oriented notation, used to describe interactions between the user and the interface [54]. This work explores the formalisation of user interactions, and inspires a way of ensuring consistency between user interface and functionalities.

We have discussed the techniques of formal methods and the formal notations used in formal methods. We have also discussed work related to the problem we want to solve: ensuring the consistency between the requirements of the interactive system and the model of the interactive system. The two formal verification techniques CIRCUS and PVSio-web contribute to, but still did not solve the problem of ensuring consistency between the front-end and back-end of the system. Currently, the related works focus on bringing the user requirements and formal models closer and getting a better understanding. The discussions reinforce our motivation to combine the use of behavioural specifications and formal methods as it provides the benefit of a better interpretation of the user requirements. We discuss this in more details next.

2.3 Combining Behaviour-driven Development and Formal Methods

2.3.1 Combining Formal and Informal Methods

Behaviour-driven development and formal methods are useful techniques for developing interactive systems, but they both have their pros and cons when it comes to safety. Combining the rigour and precision of formal specifications with the intuitive and accessible nature of informal specifications is useful in supporting the development of safety-critical interactive systems, but the major problem is we can not make sure the informal and formal specifications are consistent. The problem of ensuring consistency between the informal specification and the formal specification remains.

Research has shown that combining formal and informal methods can have positive outcomes. Wolff proposed a methodology that combines the agile development process Scrum with formal methods to assist the development of safety-critical systems [93]. The approach has two teams working in parallel where one team uses conventional development (informal) methods and the other one uses formal methods. The two teams work on their individual development iterations, then they synchronise their work at predefined milestones to ensure that they are working on the same objectives and direction. Wolff showed that the customer in his research was "especially satisfied with the early and rapid feedback provided by the formal model created".

As discussed previously, behavioural specifications have a structure in the form of "Given, When, Then" and no defined syntax, this means different writers can have very different styles of scenarios and thus these scenarios are harder

to compare and review. While formal specifications have defined syntax, semantics, and reasoning logic, it is easier to compare two formal specifications. Not having semantics and syntax means we cannot directly use our formal method techniques to prove behavioural specifications are correct or satisfied. To resolve this problem, we discuss how to integrate formal methods in particular Z notation, and informal development methods (or specifications), in particular behavioural specifications.

As discussed in [31], one way of integrating formal methods and other development methods is to use formal methods to review an existing process. One team uses certain development methods, while another separate team analyses the requirements formally and writes a formal specification for these requirements. Z has been applied successfully using this approach in [17], where a structured specification (such as an entity-relationship model, a conceptual framework used to provide a unified view of data [35]) has been translated into a Z specification. Errors can be identified and resolved during the translation. Similarly, we use the Z specification to review the behavioural specification. When there is any discrepancy between the two, we know something is amiss. We aim at finding the cause of the discrepancy and figuring out what requirement we might have missed. More about our approach will be discussed in later chapters.

When we consider using formal methods and behaviour-driven development, we must consider how we employ these two techniques. For formal methods, Bowen identified three levels to which formal methods should be employed [31]. The first level, formal specification, where specifications are written using a formal language that is more concise and less ambiguous than informal language, which is easier to reason about them. The second level, formal development

and verification, where the system to be considered is formally specified, certain properties are proved to be maintained, undesirable properties are avoided, and a refinement calculus is applied so that the abstract specification is refined into concrete representation. The third level, machine-checked proofs where correctness and well-foundedness are mechanically checked.

Naturally, we would want to apply machine-checked proofs for our safety-critical interactive system under consideration, but we must first explore the effect of combining formal methods and BDD. Thus in this work, we will start with the first level - formal specifications, and then we will look into the verification of the formal specifications.

2.3.2 Comparison Between Informal and Formal Methods

When an informal specification and a formal specification are considered together, they can not be compared directly due to different syntax, semantics, and levels of abstraction (the level of granularity). Therefore, an informal specification and a formal specification need to be at the same levels of abstraction if we want to compare them.

One way to compare informal specification and formal specification is by direct mapping using an ontology (an ontology [48] is a formal representation of the knowledge related to a domain, it can be used for terminology or concept mapping). Silva has investigated this method in [85], he developed a behavioural-based approach that supports specifying and testing user requirements in interactive systems. The ontology represents a set of concepts or objects in this domain and the relationships between these items, this will provide automated

assessments for the informal artefact. Behaviours (in the form of words) are predefined in a vocabulary (which is essentially a database) such that each behaviour will have a matching formal presentation in the vocabulary. Thus the behavioural specifications can be formalised into different artefacts such as state machines [57] (an abstract model that uses states and transitions to show changes) and task models by matching the components in the vocabulary, automatic formal testing will also be possible due to this feature. This behavioural-based ontology is built that can be used to specify the user requirements which changes the abstract level of the Behaviour-driven-development tests. However, a wide range of inconsistencies was identified when using the ontology to transform different behavioural specifications, including conflicts between specification and modelling (tasks modelled in the task model do not present the requirements specified in the user stories). Silva's approach supported the interactive system development but still could not solve the problem of ensuring that the formal model and the informal artefacts remained consistent.

Later, Silva et al. proposed a high-level domain-specific language (DSL) for specifying user requirements given as behavioural specifications (scenarios) in [86] and [87]. Silva et al. provided a vocabulary for the DSL, structural elements, and rules for writing BDD scenarios. By formulating the information provided above, Silva proposed an entity model, to "represent the actions, states, and properties of each entity that compose the system under specification". With the vocabulary, the DSL provided formalism for the behavioural specifications, while readability was maintained for non-technical readers.

The DSL defines a construct "which means" that can be used to describe how a domain behaviour (at the declarative level, such as "deposit 200 dollars")

translates into the expected interactions on the user interface (at the imperative level, such as "press deposit button and input 200"), so each declarative "Given, When, Then" step is translated into an imperative BDD scenario. By applying this approach, BDD scenarios are re-written using the DSL and the domain states and actions (Given, When, Then) are translated into interactive states and actions (Given, When, Then). By using the DSL, the abstraction level of the software specification is raised.

Different languages have different syntax and semantics. To compare a behavioural specification and a formal specification, we need to know if a term in one specification has the same meaning if this term appears in the other specification. Direct mapping is a method that establishes a one-to-one correspondence between elements of two sets or domains. At the start of our research, we investigated if we could directly map one element (which could be a word, a phrase, a sentence, etc.) of a behavioural specification to another element (which could be a predicate, an operation, or a whole schema, etc.) in a formal specification. Then a behavioural specification can be transformed into a formal observation by mapping bits of information into predicates or formal presentations. In prior work, we proposed a method for the translation of BDD specifications into first-order logic predicates [26]. In the translation, each step in a behavioural specification scenario is mapped into a first-order logic predicate. This approach inspires a way of turning scenarios into something less informal. However, after further experiments, we realised the relationship between the clauses is not represented in the predicates, and the causation information is not preserved. The transformation can only handle simple sentences such as "equal to", "add", or "exist", more complex information can not be translated. As behavioural specifications use natural language, and we are not able to cover the translation of natural language, thus this method is not practicable at the current stage.

Another way of making the comparison applicable is finding some intermediate-level formal presentation that is comparable with the formal specification, so that the informal specification can be transformed into this formal presentation. Ghazel et al. introduced a refinement method in [49] for transforming requirements written in natural language into formal requirements. Three refinement patterns (clarify, split, and modify) are established which are used under different preconditions. The three patterns are applied to the informal requirements during the refinement process, the refinement process is repeated until the refined requirements are directly formalisable. As soon as the refined requirements are considered "directly formalisable", the formalisation step is performed. The informal parts in the refined requirements are mapped into formal propositional variables or predicates and the refined requirements are transformed into CTL* (a superset of computational tree logic (CTL) and linear temporal logic (LTL)) formulas. These CTL* formulas preserve the requirement information and they can be verified. Ghazel's research brought some inspiration to formalising the artefacts, but this approach applies "Modify pattern", which means "requirements must be modified if they contain inconsistent information or errors". The engineer's expertise is imperative in this modification, which means the correctness of the result relies on the person modifying the requirements, and the problem of inconsistencies remains.

Another research on transforming informal specifications into formal representations is presented by Bowen and Reeves in [25] where they 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 is combined with finite state machines (FSA, a mathematical model of computation that can be used to represent and control execution

flow) to present the dynamic behaviours of the UI, called Presentation and Interaction Model (PIM). This approach adopts the method of directly mapping informal artefact components to formal observations.

Similarly, Haeson et al. present an approach [51] which makes use of the MuiCSer [50] process framework, that maps COMuICSer [52] storyboards into UsiXml [69] models (a model that contains task model, abstract user interface model, context model, etc.). The MuiCSer consists of five steps: Requirements and user needs, structured interactive models, low-fidelity prototypes, high-fidelity prototypes and finally the interactive system. A storyboard is defined as " a sequence of pictures of real-life situations, depicting users carrying out several activities by using devices in a certain context, presented in a narrative format." The COMuICSer storyboards are mapped into UsiXml models by deciphering each part of the storyboard into a respective part in UsiXml Model. The MuiCSer process increases the traceability (ability to track and managing changes at each stage back to its origins and forward to final implementation) and visibility of the artefacts.

The successful attempts at formalising informal artefacts including unstructured artefacts like UIs, act as a step towards the goal of integrating formal and informal methods. The research discussed above shows that such integration is possible and useful for revealing the inconsistencies between the requirements and products. Although there is currently no existing method that can solve the problem of inconsistencies, we can get closer to identifying and solving the inconsistencies we found.

2.4 Summary

In this chapter, we discussed the background knowledge of behaviour-driven development and formal methods, especially Z notation. We explained why using behaviour-driven development or formal methods individually is useful but not enough, for the high demand for correctness in safety-critical interactive systems. Using behaviour-driven development alone lacks formal verification and suffers from ambiguity in language, while using formal methods alone relies on the developer's interpretation of the requirements. Our work aims at improving correctness, by combining the two methodologies, and reducing the gap between requirements and developer interpretations.

We also discussed related research on safety-critical interactive system development, with an emphasis on comparing informal and formal methods, and further leveraging the benefits of these methods. Existing approaches attempted to detect mismatches by formalising the informal artefacts and have shown good results. Our approach is one step further on the path of identifying and solving the inconsistencies between the informal specification and the formal models. In the next chapter, we will discuss the process we propose for comparing behavioural specifications and Z specifications.

Chapter 3

Comparing Behavioural and Formal Specifications

In this chapter, we start by introducing the methodology of developing the process to compare the behavioural and Z specifications. Then we discuss how to apply the process to compare the two specifications and check for consistency in detail.

3.1 Methodology

The workflow we propose enhances the standard system development workflow by integrating a comparative analysis between behavioural specifications and formal specifications, as shown in the figure 3.1. The comparison aims to ensure the two specifications are consistent so that the final developed system meets all the requirements and satisfies both behavioural and formal specifications.

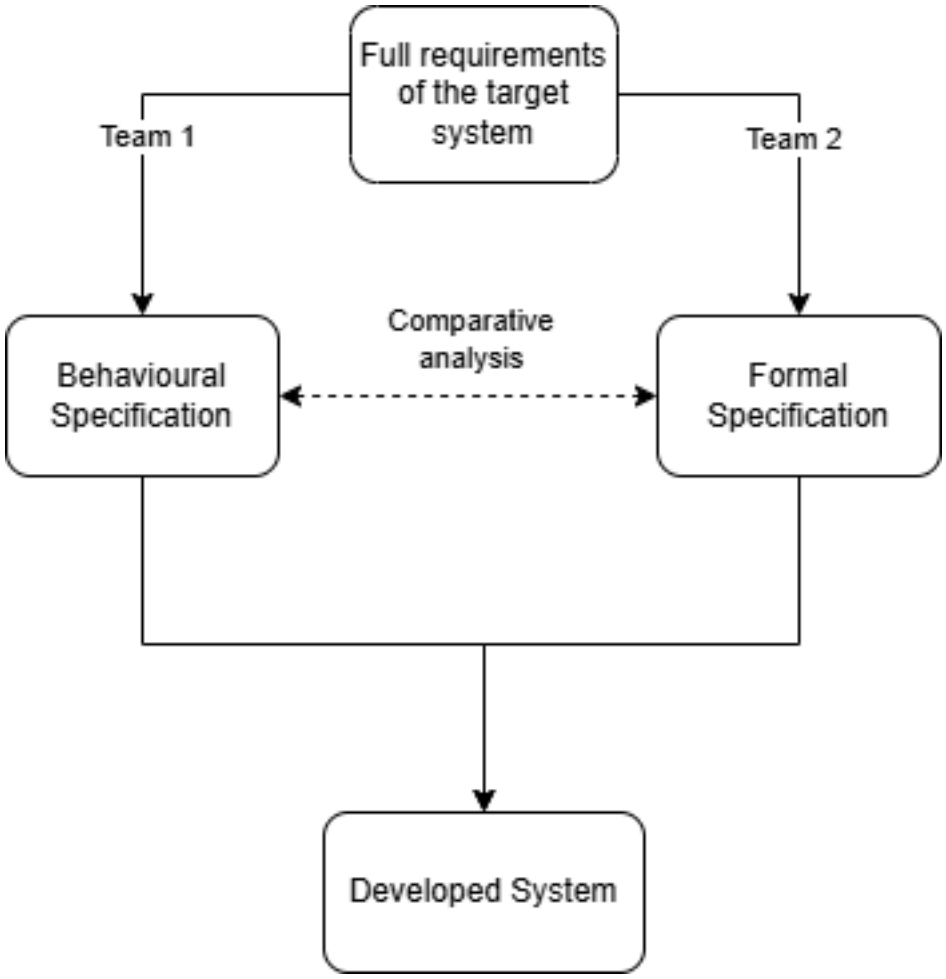


Figure 3.1: Comparative analysis step in development process

In order to perform the comparison, there are a few things needed to be considered iteratively. Firstly, the behavioural and Z specifications are not directly comparable due to different syntax, semantics, and levels of abstraction (the level of granularity), as we discussed in Chapter Two. Therefore, they need to be brought to the same level of formalism before a comparison is possible. The question is, should the behavioural specification be formalised, or should the Z specification be informalised?

To answer this question, we started by inspecting the components of Z specifications and behavioural specifications: Firstly to determine if there are existing semantics in the behavioural specifications, and secondly to find any

relationship between the behavioural and Z specifications. By inspecting the scenario in behavioural specification, we found that "Given", "When" clauses represent pre-conditions and "Then" clauses represent post-conditions, so that the "Given" and "When" conditions result in the "Then" outcome; operations in Z specification also have this characteristic. Thus, this finding suggested a way forward. Next we analysed the meaning of formal and informal, a specification is formal because it is expressed in a rigorous and unambiguous way, meaning there is only one interpretation of this specification. If the formal specification is informalised, it is difficult to maintain rigour, and ambiguities can potentially be introduced. On the other hand, an informal specification can contain ambiguities and implicit assumptions, there can be multiple ways of interpreting an informal specification. Formalising an informal specification could help eliminate these ambiguities. A formalised artefact of an informal specification is also useful for further analysis or refinement. Therefore, it is more logical to formalise the behavioural specifications, this leads to the next question: What representation are we formalising the behavioural specification into?

To answer this question, we analysed what information in the behavioural specification we needed to formalise and what formal representation can represent such information. The ideal formal representation needs to be comparable with the Z specification. We investigated several formal representations (such as FSMs and modal logic for possible worlds) and mathematical notations, in particular the mathematical logic (in this case, axiomatic set theory and first-order predicate logic) used by Z notation. We experimented with these representations by formalising some information in the behavioural specification to see if the representations are suitable to represent the required information. First-order logic emerged as a suitable candidate to transform the behavioural specifications into. We next needed to identify which parts of the

behavioural specification needed to be formalised, and which parts of the existing behavioural specification would provide that information.

To address these questions, we broke the behavioural specifications into narratives and scenarios. Then the scenarios, grouped by features, were further broken into a series of sentences and words. We experimented to see whether the words would be transformed individually or as a whole sentence, and how we could organise the words before and after transformation. By investigating the different parts of the behavioural specification we aimed to find which parts may lend themselves to the first-order logic representation (pre- and post-conditions). We identified that the "Given" and "When" conditions needed to be formalised, and the remainder (description, narrative and "Then" outcome) of the behavioural specification provides the information needed to identify and organise the conditions. Once the behavioural specification has been formalised, the next step is considered: Which parts of the Z specification are relevant and how are they related to the identified parts of the behavioural specification?

In a similar manner to our considerations of the behavioural specification, we first looked into the different parts of the Z specification to find the operations that had similar conditions and outcomes to those described in the behavioural specification. We interpreted the behavioural specification to understand the requirements it described, then we identified corresponding parts in the Z specification that reflected those requirements. Using the criteria identified in the behavioural specification, we explored the state space of the Z specification to see if the Z specification produced the same outcomes. This helped us understand how the parts of the Z specification were related to the identified parts of the behavioural specification. After we identified the relevant parts of the

Z specification and understood how they were related to the identified parts of the behavioural specification, we focussed on the next step: How to check that Z specifications and the behavioural specifications were consistent? I.e, how do we make the comparison?

To answer this question, we experimented with a few different methods. We tried to establish direct links between the identified parts in the behavioural specification and related parts in the Z specification, to ensure that each user requirement is accurately represented in the Z specification. We simulated the model in the ProB model checker, using sample initial states, to observe the behaviour of the system, so we could see if the system behaved as expected in different situations with different inputs. We used formal model checking tools to check if the formalised representation and the relevant parts of the Z specification were contradictory. We also explored whether we could refine the formalised behavioural specification into the Z specification or vice versa.

While the questions and solutions as described appear linear, they were iterated until we reached a solution that could answer all of the questions posed. This led to the development of our transformation and comparison process.

We use behavioural specifications and Z specifications with different focuses. The behavioural specifications are centered on the user requirements of the target system, which are primarily related to usability and user satisfaction. The Z specifications are centered on the computational requirements of the target system which is related to functionalities. Note that behavioural and Z specifications are derived from the same set of requirements of the system under consideration, but neither behavioural nor Z specifications necessarily describe the complete set of requirements for the system. Due to their different

focuses, however, the behavioural specification and the Z specification do not necessarily describe the same aspects or parts of the system. Each of them presents a subset of the requirements, these two artefacts combined should present the complete set of requirements for the system under consideration. When a behavioural specification and a Z specification describing the same system are considered together, the information given in the behavioural specification and the Z specification may intersect, contradict, or be disjoint.

By **intersection**, we mean the subject of a clause in the behavioural specification is the same as the subject of a statement in the Z specification. For example, if there is a clause in the behavioural specification that states "the machine has 2 sodas", while a statement in the matching Z specification states: "globally the machine has more than 1 soda", then they both describe the conditions of sodas in the machine, and this would be an **intersection**.

By **contradiction**, we mean a clause in the behavioural specification contradicts a statement in the Z specification. For example, if there is a clause in the behavioural specification that states "the machine has 2 sodas", while a statement in the matching Z specification states: "globally the machine has no less than 3 sodas", then this would be a **contradiction**.

By **disjoint**, we mean the subject of a clause in the behavioural specification is different from the subject of any statement in the Z specification. For example, if there is a clause in the behavioural specification that states "the machine has 2 sodas", while a statement in the matching Z specification states: "globally the machine has no less than 3 chocolate bars", then the clause and statement are **disjoint**.

The goal of our process is to ensure that the behavioural specification and Z specification are in agreement, in our words, the behavioural specification and Z specification are **consistent**.

3.1.1 Consistency

We start by defining what we mean by consistency. Given a behavioural specification and a Z specification describing the same system, if the requirements presented in the behavioural specification are also presented in the Z specification, and the behavioural specifications do not contradict the Z specification, then the behavioural specification is **consistent** with the Z specification. For a behavioural specification and a Z specification to be **consistent**, there are three things we need to check:

1. The intersection between the two specifications must not contradict each other.
2. Anything in the Z specification that is not in the behavioural specification must not contradict any statement in the behavioural specification.
3. Anything in the behavioural specification must also be in the Z specification.

Our approach enables a comparison of a behavioural specification with a formal specification written in Z. The goal is to check the user requirements given by the stakeholders (in the form of behavioural specification) are met in the Z specification. That is, to check for consistency, contradiction, and disjointness. As the languages used in behavioural specifications and Z specifications are different, we need a way to make them comparable, thus we need to have them at the same level of formality. We propose a process that enables the derivation of predicates from a behavioural specification, that can be used in a model-checking process of the formal specification.

If we consider the relationship between a behavioural specification and a Z specification in terms of their coverage of the requirements, there are two questions we want to investigate:

1. Is there any requirement that is present in the behavioural specification, but not in the formal specification?
2. Is there any requirement in one of the specifications that contradicts a requirement in the other specification?

For question one, if there is such a requirement, then the formal specification does not capture all the requirements in the behavioural specification. In this case, the formal specification is incomplete, and the two specifications are inconsistent (for this approach we consider completeness as a subset of consistency). For question two, if there is such a requirement, then the formal specification does not support all the requirements in the behavioural specifications. The two situations above indicate that the formal specification and the behavioural specification are inconsistent, and they need to be reconsidered. Using a vending machine as an example, we demonstrate our process, to compare the specifications of the vending machine and identify requirements in common, in order to ensure consistency.

3.2 Terminology

We begin by defining the terminology used in the process.

Essential Facts

As discussed in Chapter Two, behavioural specifications use goals to describe the intended outcome of scenarios. Essential facts describe conditions required (hence essential) to achieve certain goals within a scenario. Essential facts are not part of the goals. Within a behavioural specification, the essential facts are all the requirements in that behavioural specification. Essential facts

allow us to identify the intersection between the requirements described in the behavioural specification and the requirements described in the Z specification.

Goal-Related Operations

Within the behavioural specification, each scenario in the same feature shares a goal that needs to be achieved, achieving a goal is essentially an expected behaviour of the system. Each of these expected behaviours should have a corresponding operation in the Z specification, which results in the same outcome as the behaviour, given the same precondition. We call this a goal-related operation. The goal-related operation is not enabled (i.e. the goal can not be achieved) unless the requirements (essential facts) described in the behavioural specification have been met.

Matching Statement

A matching statement is the statement (declaration or constraint) in the Z specification, whose main object is the same as the main object of an essential fact. Each matching statement has a data type, and shows where the data types of the behaviour are described in the Z specification.

Next, we explain the process that enables us to determine consistency, as described above. The process requires two artefacts: a behavioural specification and a Z specification of the system to be developed.

3.3 Comparing the Specifications

When we check a behavioural specification and a Z specification for consistency, we are trying to ensure whether the essential facts lie within the intersection between the two artefacts. Thus we are checking if the essential facts (representing the requirements in the behavioural specification) are reflected in the Z specification, and whether the essential facts contradict the Z specification or not. In this process, we only consider positive scenarios, where the goal of

a feature is always satisfied in the outcomes. Negative scenarios, whose outcomes are failures (where the goal was not achieved), are not considered. We discuss this further in Chapter Five.

The process to determine consistency between the behavioural specification and the Z specification is shown in listing 14. If any step fails, it indicates inconsistency between the specifications, and they should be reviewed to determine which, if either, is correct. Details of the steps will be discussed in the following sections, a vending machine example is used to demonstrate the process.

Listing 14 Process of comparing behavioural and Z specifications

1. Finding essential facts
2. Identifying goal-related operations and finding matching statements
3. Analysing essential facts and forming predicates
4. Checking predicates

3.3.1 Step One Finding Essential Facts

Given a behavioural specification, that describes a behaviour under some conditions, examine all the features and identify their goals. A goal is stated in the narrative's "I want to" statement of a feature, it is what the user in this feature is trying to achieve.

Scenario of Vending Machine Example

As an example, a vending machine operation has the following feature shown in listing 15.

Listing 15 A vending machine scenario for buying a snack using money

```
Feature: Buying a snack using money
As a user
Bowen wants to buy a snack with money
So that Bowen can get a snack to eat

Scenario: Buying a cookie using money when the cookie
is in stock
Given cookie is in stock
When Bowen inserts 10 dollars
And Bowen selects cookie
Then the vending machine dispenses a cookie
```

Different features can have the same narratives and contribute to achieving the same goal. For the vending machine feature, the goal is "buy a snack with money". After identifying the goals, scenarios within the features are checked to identify all the essential facts. The essential facts are grouped by the goals of the features they relate to. Identifying the essential facts is done by checking the "Given" clause, and the "When" clause of the scenario, and extracting the conditions required. The essential facts are required to achieve the goals and they capture the intent of the writer, so identifying them is useful for understanding the requirements. Note that, if the same condition appears in two features that both contribute to the same goal, it is assumed to have the same meaning, and it only needs to be added once. This is because, within a single goal, a condition should ideally have a unified interpretation. If a condition's meaning varies within the same goal, this inconsistency suggests issues in the behavioural specification itself, warranting a review or reconsideration to ensure clarity and alignment across features. If the same condition appears in features associated with different goals, it should not be assumed to have the same meaning due to the nuances of natural language in behavioural specifications. It is possible that similar phrases may imply different meanings based on context. Therefore, each condition should be treated independently and processed to ensure the distinct requirements and intentions behind each

goal are accurately captured.

Pattern for Identifying Essential Facts For essential facts, the aim is to find nouns that are part of a declarative sentence, the condition of such nouns is most likely the essential fact, but other things that look like a requirement to be fulfilled still need to be checked. Generally, an essential fact should match one of these patterns shown in listing 16.

Listing 16 Pattern for finding essential facts

1. User does Action on Object
2. Object is acted Action
3. Action is acted
4. Object is adjective

Now we explain the patterns in detail. The first pattern is "User does Action on Object", where the essential fact is "Action on Object". For example, in the clause "the user turns on the machine", the essential fact is "turns on the machine" (with or without "the" in "the machine" being irrelevant). Here, "turns on" is the action, and "the machine" is the object. The clause can vary in tenses, such as "the user turned on the machine", or "the user has turned on the machine", but the essential fact remains "turn on the machine". Optionally, the clause can include an adjective, an adverb, or both. For example, "Bowen enters the password correctly". "enters the password correctly" is the essential fact, the action is "enters", the object is "password", and the adverb is "correctly". Another example is "the user checks the door is closed", "checks the door is closed" is the essential fact, the action is "checks", the object is "door", the adjective is "closed".

The second pattern is "Object is acted Action", where the essential fact is "Object is acted Action". For example, the clause "the coin is inserted" matches

the pattern, with “the coin is inserted” as the essential fact, “the coin” as the object, and “inserted” as the action. The clause can also be in different tenses, such as “coin was inserted”, or “coin has been inserted”, the essential fact is still “the coin is inserted”. Optionally, the clause can contain a user, an adverb, or both. For example, “the options are correctly configured by the engineer”, “options are correctly configured” is the essential fact, the object is “ options”, the adverb is “correctly”, the action is “configured”, and the user is “the engineer”.

The third pattern is “Action is acted”, where the essential fact is “Action is acted”. For example, the clause “the priming is done” matches the pattern, “priming is done” is the essential fact, and the action is “priming”. Note that, the clause can also be in different tenses, such as “the priming was done” or “the priming has been done”, the essential fact is still “priming is done”. Optionally, the clause may contain a user, an adverb, or both. For example, “the coding is correctly finished by the programmer”, where “coding is correctly finished” is the essential fact, the action is “coding”, the adverb is “correctly”, and the user is “programmer”.

The fourth pattern is “Object is adjective”, where the essential fact is “Object is adjective”. For example, “the answer is correct” matches the pattern, “answer is correct” is the essential fact, the object is “answer”, and the adjective is “correct”. The clause can be in different tenses, such as “the answer was correct”, and the essential fact remains “answer is correct”.

For the scenario shown in listing 15, we start by looking at the “Given” clause. The essential fact in the “Given” clause is “cookie is in stock” which matches the pattern “Object is acted Action” in listing 16 number 3. Next, we look at the “When clause”, the essential facts are “inserts 10 dollars” and “selects

cookie" as they match the pattern "User does Action on Object" in listing 16 number 1. A clause in a scenario may look like a requirement, but when it is the same as the goal of the scenario, it is not considered an essential fact. A clause that directly achieves the goal is not part of the conditions that must be met to achieve the goal, it is the action of achieving the goal in the user story (scenario). For the scenario shown in listing 15, if there is a clause "Bowen buys a cookie", then this clause is not an essential fact.

Note that, every essential fact identified represents one condition that needs to be satisfied to achieve the goal. Although the relationship between the essential facts is unknown, in this process, we assume each essential fact can be satisfied independently, without considering whether one requirement needs to be satisfied to enable another. We discuss this further in Chapters Four and Five.

For each goal identified, a list of assumptions is constructed for all the scenarios that contribute to the same goal, each assumption in this list is assumed to be true when we are considering any scenario that contributes to this goal. Although each scenario should only emphasise one condition that is required to achieve the scenario's goal, there are often more conditions included in a scenario. Thus the purpose of the assumption list is to include all the conditions needed for a feature. For the scenario shown in listing 15, the scenario has a goal "buy a snack with money", the assumption list for "buy a snack with money" has essential facts "cookie is in stock, inserts 10 dollars, selects cookie", these essential facts are assumed to be true whenever a scenario with the goal "buy a snack with money" is considered. All the essential facts are identified from the behavioural specification before moving on to the next step.

3.3.2 Step Two Identifying Goal-Related Operations and Finding Matching Statements

For each goal identified earlier, the 'goal-related operation' is identified for the feature. This is the operation in the specification that corresponds to the user behaviour which contributes to the goal.

To identify the goal-related operation, the Z specification is searched for the operation that has similar conditions and outcomes as the goal, this is the goal-related operation. The reason why we look for **similar** instead of exactly the **same** conditions and outcomes, is because the writers of behavioural specifications (stakeholders) and Z specifications (developers) are usually different groups of people, and may use different wordings. The different use of wordings may result in ambiguity and inconsistency, which is one of the problems we are trying to solve in this research. In the case where there is more than one suitable operation, it is necessary to consider how the two operations are combined. For example, as shown in listing 17.

Listing 17 Picking an operation

$$
\begin{array}{|l}
\hline
OperationA \\
\hline
OperationA1 \\
DeclarationB \\
DeclarationC \\
\hline
ConstraintD \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
OperationA1 \\
\hline
DeclarationE \\
DeclarationF \\
\hline
ConstraintG \\
\hline
\end{array}
$$

There are two relevant operations, A and A1. A1 is included in A, A and A1

have similar conditions and outcomes as the goal of the feature. Since A1 is part of A, A is picked as the goal-related operation, as whatever predicates (essential information) are included in A1 are also included in A. Picking A over A1 allows us to review a wider range of information. This can help with deciding whether the essential fact is in the intersection of the two specifications or not.

Identifying Goal-Related Operation For Vending Machine Example

As an example, we have the following operation for the vending machine:

Listing 18 BuyVM operation

$$
\begin{array}{|l}
\hline
BuyVM \\
Pay \\
Take \\
\hline
price(item?) = cost? \\
\hline
\end{array}
$$

Listing 19 Pay and Take operations

$$
\begin{array}{|l}
\hline
Pay \\
\Delta Money \\
cost? : \mathbb{N} \\
\hline
currentMoneyValue \geq cost? \\
currentMoneyValue' = currentMoneyValue - cost? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
Take \\
\Delta Inventory \\
item? : snack \\
\hline
item? \in inventory \\
inventory' = inventory \uplus [\![item?]\!] \\
\hline
\end{array}
$$

As shown in listing 18"BuyVM" is an operation schema that combines two operations "Pay" and "Take", they are shown in listing 19. By looking at the operations, "Pay" involves paying for the product, and "Take" involves taking

the product from the vending machine's inventory. Being a combination of "Pay" and "Take", "BuyVM" involves buying from the vending machine and gaining the selected product, thus it has a similar condition and outcome as the goal "buy a snack with money", "BuyVM" is identified as the goal-related operation.

To find out how the data types of the behaviour are presented in the Z specification, the matching statements for the essential facts need to be found. To find the matching statement, we look at each of the following in listing 20:

Listing 20 Where to find matching statement

1. Type declarations
2. System schema
3. Axiomatic definitions or predicates
4. Operation schema

First, we consider the type declaration of the Z specification. If the type declaration does not have a matching statement for the essential fact being processed, we examine the state schema and see if there is any observation that matches. If this fails, we look at the Axiomatic definitions of the Z specification and the operation schema. If a matching statement is identified in the state schema, everything in the operation schema that relates to this matching statement is also a matching statement. Note that a matching statement can be a type declaration, a declaration, or a constraint (predicate). We do not stop at the first matching statement found, we go through the whole list in listing 20 and find all the matching statements, as they are useful for reinforcing our understanding and interpretation of the essential fact. Therefore, if a matching statement has a type defined in a type declaration, that type dec-

laration is also a matching statement, as this type declaration is useful for us to understand the requirement represented in the essential fact. The locations of the matching statements should be recorded. Note that if a property is unchanged, for example "A = A'", it does not need to be marked as a matching statement. If there is no matching statement found for an essential fact, the Z specification is inconsistent with the behavioural specification.

Finding Matching Statements for Vending Machine Example

For example, if we consider the essential fact "Cookie is in stock" for the scenario in listing 15, where "Cookie" is the main object. We look for a matching statement for "Cookie" in the Z specification:

$$snack ::= cookie | chocolate | chips$$

The type declaration above has "cookie" as one of its possible values, thus "snack ::= cookie | chocolate | chips" is a matching statement for the essential fact "cookie is in stock". For the "take" operation shown in listing 19, there are two predicates:

$$item? : snack$$
$$item? \in inventory$$

From interpreting the Z specification and the previous matching statement, "item?" has the type "snack", which includes "cookie". "item?" is an element of "inventory". Therefore, the two predicates are both matching statements for the essential fact "cookie is in stock". Finding matching statements from axiomatic definitions and state schema is done in the same manner as above.

We go through the entire Z specification and find out all the matching statements for the essential facts. The matching statements in the Z specifications

are essentially the "intersection" of the behavioural specification and the Z specification, i.e. where common information is presented in both specifications. For an essential fact, the matching statements' type will be used as the type of the essential fact. The matching statements of an essential fact should have the same type, if there is more than one type, either the process is conducted incorrectly, or the specifications are incorrect and need to be reviewed. Note that, for simplifications, Z specification writers may split a complex type into simpler types. For example, instead of using a single "time" type, a model may use 'hours", "minutes", and "days". In such cases, an essential fact can adopt multiple types, without breaking the rule of only having one type for each essential fact, because these types represent the same underlying type. For example, an essential fact "the time-out limit is 1 hour and 30 minutes". If the Z specification uses types "hour" and "minutes" to represent time, then "the time-out limit is 1 hour and 30 minutes" may have both "hours" and "minutes" as its type, since these simpler types collectively represent the complicated type "time". Having multiple types for an essential fact in such cases does not affect the application or the result of the process.

Finding the type helps interpret the meaning of the essential fact. For the vending machine example, "item?" has "snack" as its type, and "inventory" is a bag of "items". Thus "snack" is the type for essential fact "cookie is in stock".

Generalisation

Once the goal-related operations have been identified and types are given to the essential facts, we next check if the scenarios need generalisation. Generalising a scenario means replacing any concrete values in the scenario with the actual meanings of these conditions.

Using the schema discussed previously in listing 18, we consider the scenario in

listing 15. The scenario has three essential facts: “cookie is in stock”, “inserts 10 dollars”, “selects cookie”. “cookie is in stock” and “selects cookie” have a concrete value “cookie”, “inserts 10 dollars” has a concrete value “10 dollars.” Using the matching statements found earlier, the two essential facts with concrete value “cookie” have type: “snack ::= cookie | chocolate | chips”, thus it is possible to replace “cookie” with a more general word “snack”. “inserts 10 dollars” has type “$\mathbb{N}$”, which is essentially the amount of money value.

After that, we interpret the intent of the writer. For “cookie”, if the intent is to buy specifically “cookie”, then this part does not need generalisation; if the intent is to buy any selected “snack” (the type itself, could be chocolate or chips, etc.) and not just “cookie”, then “cookie” can be generalised to “snack”. For “10 dollars”, if the purchase requires specifically 10 dollars, then it does not need generalisation; if the intent is to buy the item with money and the price restriction is not mentioned, or the price is a certain value “A”, then “inserts 10 dollars” can be generalised to “insert more than A dollars”, or simply “insert money”.

The intent should be described in the feature, if there is no additional description in the feature description or scenario title about the specific values, then the values are not specific and should be generalised. As shown in listing 15, the goal is to “buy a snack with money”, we see no additional information for either “cookie” except for “buy a snack” as its goal, or “10 dollars” except for “money”, we therefore generalise the scenario, as shown in listing 21.

Listing 21 Generalised scenario Buying a snack using money

```
Feature: Buying a snack using money
As a user
Bowen wants to buy a snack with money
So that Bowen can get a snack to eat

Scenario: Buying a snack using money when the snack is
in stock
Given snack is in stock
When Bowen inserts money
And Bowen selects a snack
Then the vending machine dispenses a snack
And Bowen gets a snack to eat
```

When a scenario is generalised as in the example above, we replace the concrete value in the essential facts with the range of values (type such as "snack", certain value or range such as "'enough money', 'money', or more than '10 dollars") the essential facts represent. For example, essential fact "cookie is in stock" will become "snack is in stock" and essential fact "inserts 10 dollars" will become "inserts money".

Once the goal-related operations and matching statements are identified, they can be used in the next step - to analyse the essential facts and form predicates.

3.3.3 Step Three Analysing Essential Facts and Forming Predicates

In this step, the Z specification with goal-related operations identified and the typed essential facts are used to form first-order logic predicates. Each goal has an assumption list, every essential fact in this list is considered and used to form predicates respectively. The assumption list for the generalised vending machine scenario shown in listing 21 is: "snack is in stock, inserts money, selects snack".

The meaning of each essential fact is interpreted and understood, based on reading the essential fact's scenario and its narrative. Then predicate (s) are formed based on the meaning of the essential fact as well as the constraints in the goal-related operation, each predicate will represent a requirement described in an essential fact. A predicate is a logical statement that must be true in the goal-related operation in the Z specification.

For example, the essential fact "snack is in stock" means a requirement that "there is a snack in the stock inventory", which translates to a predicate "$snack \in inventory$" which means the vending machine inventory has "snack". Similarly, the essential fact "inserts money" means a requirement that "some money is inserted", it will have a predicate "$currentMoneyValue > 0$" which means the value of the money entered is greater than 0. The essential fact "selects snack" describes a requirement "the item selected by the user is a snack", thus it will have a predicate "$item? \in snack$", which means the item selected belongs to the snack set.

All predicates of a goal will be put into the predicate list for that goal. For the vending machine example, goal "buy a snack with money" will have predicates "$snack \in inventory$, $currentMoneyValue > 0$, $item? \in snack$". These predicates are then checked against the Z specification in the next step.

3.3.4 Step Four Checking Predicates

For the predicates list of a goal, if any predicate in the list is unsatisfied, the goal-related operation related to this goal should not be enabled. Given goal A, with a list of predicates AP with predicates "A1, A2, A3", and the goal-related operation for goal A is AO. A check is done in this way: globally (i.e.

in every state), if "A1 and A2 and A3" are not satisfied (i.e. not all of them are satisfied, or say any of them is not satisfied), the operation AO should not be enabled.

The statement above translates to the following linear temporal logic formula:
$G(not(A1\&A2\&A3) => not(e(AO)))$
Using De Morgan's law, the formula is equivalent to:
$G(((notA1)or(notA2)or(notA3)) => not(e(AO)))$
The predicates will always be in the form as shown above.

ProB is the model checker and analyser introduced in Chapter Two. ProB supports the checking of LTL assertions. When an LTL formula does not pass the check, a counter-example is provided. If any of the predicates described in LTL assertions fail during model checking, it indicates that the behavioural specification and the Z specification are inconsistent. If all of the checks passed, the behavioural specification and the Z specification are consistent.

Checking Predicates for Vending Machine Example

As an example, the goal "buy a snack with money" has predicates "$snack \in inventory$, $currentMoneyValue > 0$, $item? \in snack$". The goal-related operation is "BuyVM". This information translates to the following linear temporal logic formula:
$G(((not\ \{snack \in inventory\})or(not\{currentMoneyValue > 0\})or(not\{item? \in snack\})) => not(e(BuyVM)))$
The formula is checked against the Z specification of the vending machine in ProB. The check passed and therefore no inconsistency is found between the processed behavioural specification and the Z specification.

It is important to note that, the process does not guarantee the correctness of the considered system. The process only checks for the inconsistency between the compared behavioural specification and Z specification.

3.4 Summary

In this chapter, we discussed the methodology of creating the process of comparing behavioural and Z specifications, We introduced the questions we sought to answer in the process and the methods taken to answer these questions. We have also described and demonstrated the process for comparison. In the next chapter, we will use a real-world example, which is a medical infusion pump, and demonstrate how we compare its behavioural specification and Z specification.

Chapter 4

Infusion Pump Example

In this chapter, we present a concrete example to demonstrate the application of the process for checking the consistency between a behavioural specification and a Z specification. We also discuss the difficulties and findings of applying the process. The NIKI T34 syringe driver is used as an example. The T34 is a small, portable battery-powered syringe pump, designed to deliver the contents of a syringe to a patient over a specified duration or at a given rate in millilitres per hour (ml/hr). A picture of the T34 syringe driver is shown in figure 4.1. The pump actuator drives the syringe plunger forward at a controlled rate and the rate is adjustable. A detailed operating guideline for the NIKI T34 syringe driver can be found at [15]. The T34 infusion pump is used as our example because it is a real-world safety-critical interactive system, which is what our work is aimed at. It has complex back-end behaviours and demonstrates the relevance of our work to real-world systems. It is also important to note that very few full Z specifications for safety-critical interactive systems are publicly available. Previous work has been done on modelling the T34 system by Jaidka [60] and we have access to a full Z specification for the T34 infusion pump.

The pump user needs to load the syringe onto the top of the pump. The pump

Figure 4.1: T34 infusion pump

will detect and display the size and brand of the syringe, on the display screen. The user can also select brands using the "Up" and "Down" arrow keys if the syringe brand detected is incorrect. Once the information is confirmed, press the "Yes" key to continue. The pump calculates and displays the deliverable volume in the syringe, the user can use the arrow keys to adjust the "Volume To Be Infused" (VTBI). After setting up VTBI, the user needs to press the "Yes" key again to confirm the VTBI. The LCD display will prompt the user to set the duration of the infusion (infusion means putting fluids into the patient's bloodstream), which can be set using the "Up" and "Down" arrow keys. After that, the "Yes" key needs to be pressed to confirm the duration. The Pump will calculate and display the rate required to deliver the VTBI over the infusion duration confirmed, which can also be adjusted using the "Up" and "Down" arrow keys. The pump will ask the user to confirm multiple times before the infusion can start.

4.0.1 Behavioural Specification

Here we present a partial behavioural specification that describes the intended behaviours of a T34 syringe driver. The behavioural specification is based on the instructions given in the user manual [15], we present several features for the syringe driver, such as loading the syringe into the driver and setting up the infusion. The behavioural specification is partial (in contrast to a complete set

of behavioural specifications that cover all the requirements) for the purpose of demonstrating our process, and was written specifically for the example used here. The fact that it is partial will not affect the process or the outcome of the process, as we are checking the consistency between the existing behavioural and Z specifications. We now explain each feature in detail.

As shown in listing 22, the scenario describes the situation where the syringe is loaded into the driver. Note that if there is only one scenario in a feature, this scenario can have the same title as the feature, it depends on the intent of the writer of the behavioural specifications.

Listing 22 Feature for loading the syringe into the driver

```
Feature: Loading the syringe into the driver
As a caregiver
Bowen wants to load the syringe into the driver
So that Bowen can set up the infusion

Scenario: Loading the syringe into the driver
Given Bowen has checked the battery is full
When Bowen loads the syringe into the pump
And the syringe has been correctly identified
And Bowen has confirmed the syringe
Then the syringe is loaded
And Bowen can set up an infusion
```

As shown in listing 23, the feature describes the scenarios of setting up the infusion. There are two scenarios in the feature. Note that different scenarios can have identical conditions. As discussed in Chapter Three Step One, for the same feature, identical conditions will only be considered once, more about this will be demonstrated later.

Listing 23 Feature for setting up the infusion

```
Feature: Setting up the infusion
As a caregiver
Bowen wants to set up the infusion
So that Bowen can start giving medication to the
patient

Scenario: Setting up the infusion with the program lock
Given the syringe has been identified
And Bowen has confirmed the syringe
When Bowen activates the program lock
Then the infusion is ready to start

Scenario: Setting up the infusion with the parameters
Given the syringe has been identified
And Bowen has confirmed the syringe
When Bowen sets the VTBI to 10 ml
And Bowen sets the duration of infusion to 1 hour and
30 minutes
And Bowen confirms the infusion rate
Then the infusion is ready to start
```

The other two features for locking the program and pausing the infusion as shown in listing 24 and 25 are trivial and do not require further explanation.

Listing 24 Feature for locking the program

```
Feature: Locking the program
As a caregiver
Bowen wants to lock the program
So that Bowen can prevent accidental changes

Scenario: Locking the program
Given the infusion rate is correctly set up
When Bowen activates the program lock
Then the program is locked
And no changes can be made to the program
```

Listing 25 Feature for pausing the infusion

```
Feature: Pausing the infusion
As a caregiver
Bowen wants to pause the infusion
So that the settings can be adjusted

Scenario: Pausing the infusion
Given the infusion is running
When Bowen pauses the infusion
Then the infusion is safely paused
And the settings can be adjusted
```

4.0.2 Z Specification

The complete Z specification of the Niki T34 infusion pump is given in appendix A. The original T34 specification is derived from Jaidka's work [60] and has been modified in this work for demonstration purposes. Operation names in the Z specification were modified so the parameter and operations names are consistent with the T34 user guide. Parameter and operation names in the Z specification are expected to be similar to the behavioural specifications, as the two specifications are derived from the same set of requirements. If the names are not similar, the two specifications should be reviewed to resolve this issue.

4.1 Applying the Process

Now we demonstrate how to compare the behavioural and Z specifications described to check consistency, in a step-wise manner.

4.1.1 Step One Finding Essential Facts

To start, examine all of the features and identify their goals. Our example has four features that produce the following goals as shown in listing 26.

Listing 26 Goals of the features for the T34

1. Load the syringe into the driver
2. Set up the infusion
3. Lock the program
4. Pause the infusion

After identifying the goals, all scenarios are checked to identify the essential facts, following the patterns described in Chapter Three Step One. The essential facts are grouped by their respective goals.

Load the Syringe Into the Driver

There is only one scenario with the goal "load the syringe into the driver".

Scenario Load the Syringe Into the Driver

Given From "Given", there is a clause "Bowen has checked the battery is full". In this clause, "Bowen" is the user, and "has checked the battery is full" is the requirement, it matches the pattern "User does Action on Object", thus the essential fact "check battery is full" (changed from present perfect tense to present tense). "check battery is full" is added to the list of assumptions for the goal "Load the syringe into the driver".

When From "When", there are three clauses. "Bowen loads the syringe into the pump" is the "action" of achieving the goal, the goal would be achieved if this requirement is achieved, but it can not be achieved unless all the other requirements have been achieved. Hence "Bowen loads the syringe into the pump" is not an essential fact. "the syringe has been correctly identified" matches the pattern "Object is acted Action", so an essential fact "the syringe is correctly identified" is added to the assumption list. "Bowen has confirmed the syringe" matches the pattern "User does Action on Object", so an essential fact "confirm the syringe" is added to the list of assumptions.

Having checked the Given clause and When clauses for this scenario, the assumption list for this goal now has three essential facts:“check battery is full”, “the syringe is correctly identified” and “confirm the syringe”. Since there is only one scenario, this step for goal “Load the syringe into the driver” is completed, and the result is shown in listing 27. Next, the scenarios with the goal “set up the infusion” are reviewed.

Listing 27 Assumption list for T34 after Goal One

1. Load the syringe into the driver
   1.1. check battery is full
   1.2. the syringe is correctly identified
   1.3. confirm the syringe
2. Set up the infusion
3. Lock the program
4. Pause the infusion

Set Up the Infusion

There are two scenarios with the goal “set up the infusion”.

Scenario Setting Up the Infusion with the Program Lock

Given “the syringe has been identified” is a requirement that matches the pattern “Object is acted Action”, the essential fact “the syringe is identified” is added to the assumption list for goal “set up the infusion”. “Bowen has confirmed the syringe” matches the pattern “User does Action on Object”, an essential fact “confirm the syringe” is added to the assumption list.

When “Bowen activates the program lock” matches the pattern “User does Action on Object”, so the essential fact is “activate the program lock” and is added to the assumption list.

Now the scenario “setting up the infusion with the program lock” is checked, the assumption list for “set up the infusion” has three essential facts: “the syringe is identified”, “confirm the syringe”, “activate the program lock”. Now the next scenario with the goal “set up the infusion” is checked.

Scenario Setting Up the Infusion with the Parameters

Given The “Given” clauses in “Setting up the infusion with the parameters” are identical to the “Given” clauses in “Setting up the infusion with the program lock”. The essential facts extracted are already added to the assumption list for “set up the infusion” and are not added again.

When “Bowen sets the VTBI to 10m” matches the pattern “User does Action on Object”, so the essential fact is “set the VTBI to 10 ml” and is added to the assumption list. “Bowen sets the duration of infusion to 1 hour and 30 minutes” matches the pattern “User does Action on Object”, so the essential fact “set the duration of infusion to 1 hour and 30 minutes” is added to the assumption list. “Bowen confirms the infusion rate” matches the pattern “User does Action on Object”, so the essential fact is “confirm the infusion rate” and is added to the assumption list.

The scenarios with the goal “set up the infusion” are now checked and have six essential facts “the syringe is identified”, “confirm the syringe”, “activate the program lock”, “set the VTBI to 10 ml”, “set the duration of infusion to 1 hour and 30 minutes” and “confirm the infusion rate” in the assumption list. The result is shown in listing 28. Next, the scenario with the goal “lock the program” is reviewed.

Listing 28 Assumption list for T34 after Goal Two

1. Load the syringe into the driver
   1.1. check battery is full
   1.2. the syringe is correctly identified
   1.3. confirm the syringe
2. Set up the infusion
   2.1. the syringe is identified
   2.2. confirm the syringe
   2.3. activate the program lock
   2.4. set the VTBI to 10 ml
   2.5. set the duration of infusion to 1 hour and 30 minutes
   2.6. confirm the infusion rate
3. Lock the program
4. Pause the infusion

Lock the Program

There is only one scenario with the goal "lock the program".

Scenario Locking the Program

Given There is only one clause in "Given", "the infusion rate is correctly set up", it matches the pattern "Object is adjective", so the essential fact "the infusion rate is correctly set up" is added to the assumption list for "lock the program".

When There is only one clause "Bowen activates the program lock", which is the action of achieving the goal "lock the program", thus it is not a condition. It is worth noting that "Bowen activates the program lock" appeared in Section 4.1.1 too, but they are treated differently because "Bowen activates the program lock" in that scenario is a condition, while in the "Locking the program" scenario "Bowen activates the program lock" is an action to achieve the goal.

The scenario with the goal "lock the program" is now checked and has one

essential fact “the infusion rate is correctly set up” in its assumption list. The result is shown in listing 29. Next, the scenario with the goal “pause the infusion” is reviewed.

Listing 29 Assumption list for T34 after Goal Three

1. Load the syringe into the driver
   1.1. check battery is full
   1.2. the syringe is correctly identified
   1.3. confirm the syringe
2. Set up the infusion
   2.1. the syringe is identified
   2.2. confirm the syringe
   2.3. activate the program lock
   2.4. set the VTBI to 10 ml
   2.5. set the duration of infusion to 1 hour and 30 minutes
   2.6. confirm the infusion rate
3. Lock the program
   3.1. the infusion rate is correctly set up
4. Pause the infusion

Pause the infusion

There is only one scenario with the goal “pause the infusion”.

Scenario Pausing the Infusion

Given There is only one clause in “Given”, “the infusion is running”, it matches the pattern “Object is adjective”, so the essential fact “the infusion is running” is added to the assumption list for “pause the infusion”.

When There is only one clause “Bowen pauses the infusion”, which is the action of achieving the goal “pause the infusion”, thus it is not considered an essential fact.

The scenario with the goal “pause the infusion” is now checked and has one essential fact “the infusion is running” in its assumption list. The result is

shown in listing 30. Step One is now finished for the T34 behavioural specification, the next step is to identify goal-related operations and find matching statements.

Listing 30 Assumption list for T34 after Goal Four

1. Load the syringe into the driver
   1.1. check battery is full
   1.2. the syringe is correctly identified
   1.3. confirm the syringe
2. Set up the infusion
   2.1. the syringe is identified
   2.2. confirm the syringe
   2.3. activate the program lock
   2.4. set the VTBI to 10 ml
   2.5. set the duration of infusion to 1 hour and 30 minutes
   2.6. confirm the infusion rate
3. Lock the program
   3.1. the infusion rate is correctly set up
4. Pause the infusion
   4.1. the infusion is running

4.1.2 Step Two Identifying Goal-Related Operations and Finding Matching Statements

To decide whether an essential fact is in the intersection between the requirements described in the behavioural specification and the requirements described in the Z specification, the correspondence between the two specifications needs to be found, this is done by identifying two types of artefacts in the Z specification - goal-related operations and matching statements. In this step, the Z specification is reviewed and a goal-related operation is identified for each goal. After that, all the matching statements are identified in the Z specification for each goal. If possible, the scenarios will be generalised after the matching statements are identified.

Load the Syringe Into the Driver

The operation "PreLoading" as shown in listing 31 is identified. "PreLoading" checks that if syringe (SyringeOK), barrel (BarrelOK), collar (CollarOK) and plunger (PlungerOK) are not ready (represented as "no", which means the syringe is not loaded into the driver yet), the parameters will be set up accordingly as the syringe is placed into the pump. This operation is responsible for pre-loading the syringe into the driver, thus the goal "load the syringe into the driver" is achieved after "PreLoading" which makes "PreLoading" a goal-related operation.

Listing 31 Preloading operation

$$
\begin{array}{|l}
\hline
PreLoading \\
\Delta T34 \\
brand? : SyringeBrand \\
syringeSize? : millilitres \\
volumeRemaining? : millilitres \\
plungerPosition? : millilitres \\
VTBI? : millilitres \\
\hline
SyringeOK = no \\
BarrelOK = CollarOK = PlungerOK = no \\
Battery' = Battery \\
KeyPadLocked' = KeyPadLocked \\
ProgramLocked' = ProgramLocked \\
TechMenuLocked' = TechMenuLocked \\
Brand' = brand? \\
SyringeSize' = syringeSize? \\
VolumeRemaining' = volumeRemaining? \\
PlungerPosition' = plungerPosition? \\
SyringeOK' = SyringeOK \\
BarrelOK' = BarrelOK \\
CollarOK' = CollarOK \\
PlungerOK' = PlungerOK \\
SystemReady' = SystemReady \\
VTBI' = VTBI? \\
Hours' = 24 \\
Minutes' = 0 \\
Minutes > 0 \lor Hours > 0 \\
InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes) \\
\hline
\end{array}
$$

Next we identify matching statements for the essential facts in the assumption list: "check battery is full", "the syringe is correctly identified", and "confirm the syringe".

Check Battery is Full

There is no matching statement in the type declaration. The system schema "T34" is shown in listing 32, it is the system schema of the T34 infusion pump that describes all the observations (properties) of the pump, such as battery status and keypad locking status. In "T34", the matching statement is "$Battery : YesNo$" as indicated by the blue text, it is the status of the battery which indicates whether there is enough battery. By having "$Battery : YesNo$" as a matching statement, its type "$YesNo ::= yes|no$" in the type declaration is also considered a matching statement. In the axiomatic definition, there is no matching statement for the essential fact. In the operation schemas, everything that relates to "$Battery : YesNo$" are marked as matching statements, in this case, there is "$Battery = yes$" in the "Init" schema.

Listing 32 T34 system schema

$$
\begin{array}{|l}
\hline
T34 \\
\hline
Battery : YesNo \\
KeyPadLocked : YesNo \\
ProgramLocked : YesNo \\
TechMenuLocked : YesNo \\
Brand : SyringeBrand \\
SyringeSize : millilitres \\
VolumeRemaining : millilitres \\
PlungerPosition : millilitres \\
SyringeOK : YesNo \\
BarrelOK, CollarOK, PlungerOK : YesNo \\
SystemReady : YesNo \\
VTBI : millilitres \\
Hours : hours \\
Minutes : minutes \\
InfusionRate : millilitresperhour \\
\hline
\end{array}
$$

The matching statements list for essential fact "check battery is full" contains three matching statements : "$Battery : YesNo$, $YesNo ::= yes|no$", "$Battery = yes$".

The Syringe is Correctly Identified

There is no suitable matching statement in the type declarations as nothing shows whether the syringe is identified. In the system schema T34, as shown in listing 32, "$Brand : SyringeBrand$" is highlighted in orange, and it is the brand of the syringe identified, thus it is considered a matching observation. Therefore, the type "$SyringeBrand ::= BDPlastipak|Terumo$" in the type declaration is also a matching observation. In the axiomatic definition, there is no matching statement for the essential fact. In the operation schema, operation "Init" has a matching statement "$Brand = BDPlastipak$" which initialises the brand of the syringe. Operation "Preloading", as shown in listing 31, has "$brand? : SyringeBrand$" as a matching statement, is it indicates the brand identified by the pump. Everything that relates to "$Brand : SyringeBrand$" is also marked as a matching statement. The type of essential fact "the syringe is correctly identified" is therefore "SyringeBrand". The matching statements list for essential fact "the syringe is correctly identified" contains four matching statements : "$Brand : SyringeBrand$", "$SyringeBrand ::= BDPlastipak|Terumo$", "$Brand = BDPlastipak$", "$brand? : SyringeBrand$".

Confirm the Syringe

There is no suitable matching statement in the type declarations as nothing shows the meaning of the status of the syringe. In the system schema T34, as shown in listing 32, "$SyringeOK : YesNo$" is highlighted in red and is the status of the syringe being OK or not, which indicates whether the syringe is confirmed, thus "$SyringeOK : YesNo$" is a matching statement. The type "$YesNo ::= yes|no$" of "$SyringeOK : YesNo$" in the type declaration is also marked as a matching statement. In the axiomatic definition, there

is no matching statement for the essential fact. In the operation schema, we mark every matching statement, in this case, there are "$SyringeOk = yes$", "$SyringeOk = no$", "$SyringeOk' = no$" and "$SyringeOk' = yes$". The type of the essential fact is therefore "YesNo". The matching statements list of essential facts "confirm the syringe" contains six matching statements (if a matching statement appears multiple times we only use them once, but their locations need to be recorded): "$SyringeOK : YesNo$", "$YesNo ::= yes|no$", "$SyringeOk = yes$", "$SyringeOk = no$", "$SyringeOk' = no$" and "$SyringeOk' = yes$". The locations of the matching statements in the Z specification are recorded, these locations can help keep track of the snippets in Z that are related to the information represented in the behavioural specification. For example, "$SyringeOk' = no$" indicates "SyringeOk" is being changed in certain parts of the Z specification, which may affect whether the essential fact "confirm the syringe" is satisfied.

Whether generalisation is needed is now checked for the feature. There are no concrete values and the essential facts represent conditions "check battery is full", "the syringe is correctly identified", and "confirm the syringe", thus generalisation is not needed. With goal-related operation and matching statements identified, Step Two for the goal "load the syringe into the driver" is now complete.

Set Up the Infusion

After going through the Z operations, the goal-related operation "ReadyToInfuse " as shown in listing 33, is found as it is the operation for indicating the infusion set-up is ready. "ReadyToInfuse" represents that if all statuses are ready, such as the syringe is loaded and the battery is ready, the system will switch to "Ready to infuse" mode, which means "SystemReady" property is set to "yes" after the operation.

Listing 33 ReadyToInfuse operation

$$
\begin{array}{|l}
\hline
ReadyToInfuse \\
\Delta T34 \\
\hline
SyringeOK = yes \\
BarrelOK = CollarOK = PlungerOK = yes \\
Battery' = Battery \\
KeyPadLocked' = KeyPadLocked \\
ProgramLocked = yes \\
ProgramLocked' = ProgramLocked \\
TechMenuLocked' = TechMenuLocked \\
Brand' = Brand \\
SyringeSize' = SyringeSize \\
VolumeRemaining' = VolumeRemaining \\
PlungerPosition' = PlungerPosition \\
SyringeOK' = SyringeOK \\
BarrelOK' = BarrelOK \\
CollarOK' = CollarOK \\
PlungerOK' = PlungerOK \\
SystemReady = no \\
SystemReady' = yes \\
VTBI' = VTBI \\
Hours' = Hours \\
Minutes' = Minutes \\
InfusionRate' = InfusionRate \\
\hline
\end{array}
$$

For the essential facts of the goal: "set up the infusion": "the syringe is identified", "confirm the syringe", "activate the program lock", "set the VTBI to 10 ml", "set the duration of infusion to 1 hour and 30 minutes", "confirm the infusion rate", we look for matching statements in the Z specification. For simplification, the matching statements and type of essential facts are listed in listings 34 and 35. Certain matching statements are shared by the essential facts, for example, "$InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes)$" is a matching statement for three essential facts "set the VTBI to 10 ml", "set the duration of infusion to 1 hour and 30 minutes" and "confirm the infusion rate". Sharing a matching statement could indicate these essential facts are more closely related than the other essential facts, potentially indicating dependency which affects whether the other essential facts can be satisfied.

Listing 34 Step Two for set up the infusion - Part One

1. the syringe is identified
   1.1. Type: SyringeBrand
   1.2. Matching statements:
      1.2.1. Brand : SyringeBrand
      1.2.2. SyringeBrand ::= BDPlastipak | Terumo
      1.2.3. Brand = BDPlastipak
      1.2.4. brand? : SyringeBrand
2. confirm the syringe
   2.1. Type: YesNo
   2.2. Matching statements:
      2.2.1. SyringeOK : YesNo
      2.2.2. YesNo::= yes | no
      2.2.3. SyringeOk = yes
      2.2.4. SyringeOk = no
      2.2.5. $SyringeOk' =$ no
      2.2.6. $SyringeOk' =$ yes
3. activate the program lock
   3.1. Type: YesNo
   3.2. Matching statements:
      3.2.1. ProgramLocked : YesNo
      3.2.2. YesNo::= yes | no
      3.2.3. ProgramLocked = no
      3.2.4. ProgramLocked = yes
      3.2.5. ProgramLocked = no ⇒ ProgramLocked' = yes
      3.2.6. ProgramLocked = yes ⇒ ProgramLocked' = no
4. set the VTBI to 10 ml
   4.1. Type: millilitres
   4.2. Matching statements:
      4.2.1. VTBI : millilitres
      4.2.2. $millilitres : \mathbb{P}\,\mathbb{N}$
      4.2.3. $millilitres = 0\,..\,100$
      4.2.4. VTBI? : millilitres
      4.2.5. $VTBI = 1$
      4.2.6. $VTBI \leq SyringeSize$
      4.2.7. $VTBI > 0$
      4.2.8. $VTBI < 100$
      4.2.9. $VTBI' = VTBI + 1$
      4.2.10. $VTBI' = VTBI - 1$
      4.2.11. $VTBI' = VTBI?$
      4.2.12. $VTBI > 1$
      4.2.13. $VolumeRemaining' = VTBI'$
      4.2.14. $InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes)$

While each essential fact is treated as an independent condition that can be satisfied individually, recognising shared matching statements helps developers stay attentive to these interconnected essential facts. This awareness is useful in later steps of the process, as it facilitates a deeper understanding of the conditions each essential fact represents. By noting these shared statements, developers gain better insight into the connections and relationships between conditions, providing more accurate interpretations and potentially uncovering subtle dependencies that might otherwise be missed. It is worth noting that "set the duration of infusion to 1 hour and 30 minutes" has two types: "hours" and "minutes". As we discussed in Chapter Three, these two types both represent "time" while being different units, and they can be transformed into each other. Thus it is justified for "set the duration of infusion to 1 hour and 30 minutes" to adopt both "hours" and "minutes" as its types.

Listing 35 Step Two for set up the infusion - Part Two

5. set the duration of infusion to 1 hour and 30 minutes
   5.1. Type: hours, minutes
   5.2. Matching statements
      5.2.1. Hours : hours
      5.2.2. Minutes : minutes
      5.2.3. $hours : \mathbb{P}\,\mathbb{N}$
      5.2.4. $minutes : \mathbb{P}\,\mathbb{N}$
      5.2.5. $hours = 0\,..\,24$
      5.2.6. $minutes = 0\,..\,59$
      5.2.7. $Hours = 1$
      5.2.8. $Minutes = 1$
      5.2.9. $Hours > 0 \lor Minutes > 0$
      5.2.10. $(Minutes < 59 \land Hours < 24) \Rightarrow (Minutes' = Minutes + 1 \land Hours' = Hours)$
      5.2.11. $(Minutes = 59 \land Hours < 24) \Rightarrow (Minutes' = 0 \land Hours' = Hours + 1)$
      5.2.12. $(Minutes = 0 \land Hours = 24) \Rightarrow (Hours' = Hours \land Minutes' = Minutes)$
      5.2.13. $(Minutes = 0 \land Hours = 0) \Rightarrow (Hours' = Hours \land Minutes' = Minutes)$
      5.2.14. $(Minutes > 0) \Rightarrow (Minutes' = Minutes - 1 \land Hours' = Hours)$
      5.2.15. $(Minutes = 0 \land Hours > 0) \Rightarrow (Minutes' = 59 \land Hours' = Hours - 1)$
      5.2.16. $InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes)$
      5.2.17. $Hours' = (((Hours * 3600) + (Minutes * 60)) - 1) \div 3600$
      5.2.18. $Minutes' = ((((Hours * 3600) + (Minutes * 60)) - 1) \bmod 3600) \div 60$
6. confirm the infusion rate
   6.1. Type: millilitresperhour
   6.2. Matching statements
      6.2.1. InfusionRate : millilitresperhour
      6.2.2. $millilitresperhour : \mathbb{P}\,\mathbb{N}$
      6.2.3. $millilitresperhour = 0\,..\,100$
      6.2.4. InfusionRate $= 0$
      6.2.5. InfusionRate > 0
      6.2.6. $InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes)$
      6.2.7. $VolumeRemaining' = ((10000 * VolumeRemaining) - (10000 * InfusionRate \div 3600)) \div 10000$

For essential fact "confirm the infusion rate", only the rate, the rate's type, and the calculation of the rate are taken as matching statements, as shown in listing 35. Although the infusion rate is calculated using "VTBI" and infusion time (hours and minutes), they are not considered matching statements, because the value changes in these properties do not affect the actual calculation formula of "Infusion Rate".

Whether generalisation is needed is checked after identifying the matching statements. The essential facts "set the VTBI to 10 ml" and "set the duration of infusion to 1 hour and 30 minutes" contain concrete values "10 ml" and "1 hour and 30 minutes". We check the scenario that contains these essential facts and the feature's description, as shown in listing 36.

Listing 36 Setting up the infusion with the parameters

```
Feature: Setting up the infusion
As a caregiver
Bowen wants to set up the infusion
So that Bowen can start giving medication to the
patient

Scenario: Setting up the infusion with the parameters
Given the syringe has been identified
And Bowen has confirmed the syringe
When Bowen sets the VTBI to 10 ml
And Bowen sets the duration of infusion to 1 hour and
30 minutes
And Bowen confirms the infusion rate
Then the infusion is ready to start
```

The goal is to "set up the infusion", which is the same as the feature title. The scenario title is "setting up the infusion with the parameters", which does not specifically require "10 ml" or "1 hour and 30 minutes". Therefore, we conclude that the behavioural specification writer is not specifically looking for "10 ml" of VTBI and "1 hour and 30 minutes" of infusion time, and the scenario should be generalised. "10 ml" and "1 hour and 30 minutes" represent a certain volume and time, they can be generalised using their types "millilitres", "hours" and "minutes" (hours and minutes combined are time). The generalised scenario is shown in listing 37.

Listing 37 Generalised setting up the infusion with the parameters

```
Feature: Setting up the infusion
As a caregiver
Bowen wants to set up the infusion
So that Bowen can start giving medication to the
patient

Scenario: Setting up the infusion with the parameters
Given the syringe has been identified
And Bowen has confirmed the syringe
When Bowen sets the VTBI in millilitres
And Bowen sets the duration of infusion in hours and
minutes
And Bowen confirms the infusion rate
Then the infusion is ready to start
```

The two essential facts "set the VTBI to 10 ml" and "set the duration of infusion to 1 hour and 30 minutes" now become "set the VTBI in millilitres" and "set the duration of infusion in hours and minutes", as shown in listing 38. The point of the generalised scenario is to remind the readers that, irrespective of the actual values, the value "VTBI" needs to be set (in "millilitres"), and the value "duration of infusion" needs to be set (in "hours" and "minutes"), for the goal to be achievable. The other essential facts do not require generalisation, Step Two for the goal "set up the infusion" is complete.

Listing 38 Generalised assumption list for set up the infusion

1. the syringe is identified
2. confirm the syringe
3. activate the program lock
4. set the VTBI in millilitres
5. set the duration of infusion in hours and minutes
6. confirm the infusion rate

Lock the Program

The operation "ProgramLock" as shown in listing 39 is identified as the goal-related operation, this is the operation that locks or unlocks the program. In "ProgramLock", if everything (syringe, barrel etc.) is ready, the status of

"ProgramLock" will be switched to "yes" if it was "no", or be switched to "no" if it was "yes".

Listing 39 ProgramLock operation

$$
\begin{array}{|l}
\hline
ProgramLock \\
\Delta T34 \\
\hline
SyringeOK = yes \\
BarrelOK = CollarOK = PlungerOK = yes \\
Battery' = Battery \\
KeyPadLocked' = KeyPadLocked \\
ProgramLocked = no \Rightarrow ProgramLocked' = yes \\
ProgramLocked = yes \Rightarrow ProgramLocked' = no \\
TechMenuLocked' = TechMenuLocked \\
Brand' = Brand \\
SyringeSize' = SyringeSize \\
VolumeRemaining' = VTBI' \\
PlungerPosition' = PlungerPosition \\
SyringeOK' = SyringeOK \\
BarrelOK' = BarrelOK \\
CollarOK' = CollarOK \\
PlungerOK' = PlungerOK \\
SystemReady' = SystemReady \\
VTBI' = VTBI \\
Hours' = Hours \\
Minutes' = Minutes \\
InfusionRate' = InfusionRate \\
VTBI \leq SyringeSize \\
Hours > 0 \lor Minutes > 0 \\
VTBI > 0 \\
PlungerPosition > 0 \\
VolumeRemaining > 0 \\
InfusionRate > 0 \\
\hline
\end{array}
$$

There is only one essential fact "the infusion rate is correctly set up" for this goal. The type and matching statements are identified for this essential fact, the result is shown in listing 40. There is no concrete value in the scenario, thus generalisation is unnecessary. Step Two for the goal "lock the program" is now complete.

Listing 40 Step Two for lock the program

1. Type: YesNo
2. Matching statements:
   2.1. ProgramLocked : YesNo
   2.2. YesNo ::= yes | no
   2.3. ProgramLocked = no
   2.4. $ProgramLocked = no \Rightarrow ProgramLocked' = yes$
   2.5. $ProgramLocked = yes \Rightarrow ProgramLocked' = no$
   2.6. ProgramLocked = yes

Pause the Infusion

The goal-related operation identified for goal "pause the infusion" is "PauseInfusion", as shown in listing 41, this is the operation for pausing the infusion. In "PauseInfusion", if everything is ready and the infusion has been started or is ready to start (which are functionally equivalent in the Z specification), the "SystemReady" status will be switched to "no", meaning the infusion can no longer start.

Listing 41 PauseInfusion operation

$PauseInfusion$

$\Delta T34$

$SyringeOK = yes$
$BarrelOK = CollarOK = PlungerOK = yes$
$Battery' = Battery$
$KeyPadLocked' = KeyPadLocked$
$ProgramLocked = yes$
$ProgramLocked' = ProgramLocked$
$TechMenuLocked' = TechMenuLocked$
$Brand' = Brand$
$SyringeSize' = SyringeSize$
$VolumeRemaining' = VolumeRemaining$
$PlungerPosition' = PlungerPosition$
$SyringeOK' = SyringeOK$
$BarrelOK' = BarrelOK$
$CollarOK' = CollarOK$
$PlungerOK' = PlungerOK$
$SystemReady = yes$
$SystemReady' = no$
$VTBI' = VTBI$
$Hours' = Hours$
$Minutes' = Minutes$
$InfusionRate' = InfusionRate$

There is only one essential fact "the infusion is running" for "pause the infusion". The type and matching statements are identified for this essential fact, the result is shown in listing 42. There is no concrete value in the scenario, thus generalisation is unnecessary. Step Two for the goal "pause the infusion" is now complete.

Listing 42 Step Two for pause the infusion

1. Type: YesNo
2. Matching statements:
   2.1. SystemReady : YesNo
   2.2. YesNo ::= yes | no
   2.3. SystemReady = yes
   2.4. $SystemReady' = no$

Step Two is now complete for the behavioural specification. Before we move on

to Step Three of forming predicates, we first discuss observations and findings that emerged during this stage of the process.

Observations

Load the Syringe Into the Driver

For the essential fact "the syringe is correctly identified", "$SyringeBrand ::= BDPlastipak|Terumo$" is picked as the type of the essential fact, which means based on existing information the essential fact is interpreted as "the syringe (brand) is correctly identified". Although alternatively the essential fact can be interpreted to encompass multiple parameters (including Brand, SyringeSize, VolumeRemaining, etc.), this would breach the rule that the essential fact should have only one type, thus we pick "SyringeBrand" as its type and decide that this essential fact describes the syringe brand. If this interpretation is erroneous, subsequent steps in the process will uncover it. This essential fact demonstrates the ambiguity that lies within the behavioural specification, integrating the Z specification can help clarify this ambiguity.

Set Up the Infusion

The matching statement "$InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes)$" shows that the infusion rate is calculated based on "VTBI", "Hours", and "Minutes", which are set up as suggested in the essential facts "set the VTBI in millilitres" and "set the duration of infusion in hours and minutes". This implies that achieving "confirm the infusion rate" may depend on "set the VTBI in millilitres" and "set the duration of infusion in hours and minutes" to be achieved first. However, due to the Z specification's non-procedural nature, dependencies are not explicitly tracked, meaning other operations may set these variables. As long as the pre-conditions are met, the goal-related operation "ReadyToInfuse" should be enabled. While the behavioural specification implies a sequential setup, this dependency between the three essential facts is not explicitly stated in the behavioural specification, and we do not

have enough information to confirm such a dependency. Therefore, our approach acknowledges potential relationships between essential facts, but they are treated as independent, which can be satisfied separately.

The goal-related operation for "set up the infusion" is "ReadyToInfuse", which is also the goal-related operation for starting the infusion. The behavioural specifications describe "set up the infusion" and "starting the infusion" separately, whereas in the Z these are combined. In the T34 Z specification, "being able to start infusion" is defined as equivalent to "completion of setting up the infusion". Here, "setting up the infusion" involves meeting a set of predicate conditions rather than completing sequential operations as in procedural programming. This means that once the predicates are satisfied, the infusion process can begin, without requiring a step-by-step sequence to be completed explicitly. Behavioural specification can sometimes describe in a procedural way that a sequence of actions is performed, due to the natural language it uses. While Z notation is not imperative, it focuses on system conditions, not steps, to define a state, conditions like "identify syringe" can be captured as multiple properties (identify VTBI, syringe brand etc.) rather than sequential actions. A behaviour specification is supposed to only describe one condition in a clause, but sometimes such a condition (identify syringe) could be complicated and is modelled as multiple properties in a Z specification.

The "$SyringeOK : YesNo$" matching statement that matches essential fact "confirm the syringe", appeared in both "load the syringe into the driver" and "set up the infusion". It may seem redundant that the process is repeated for "set up the infusion" when "$SyringeOK : YesNo$" has already been identified in "load the syringe into the driver", but such repetition is necessary. In Z, a statement's name guarantees a consistent type across uses, while behavioural

specifications, shaped by natural language, may vary. For example, "time" in one scenario could mean "hour" while in another scenario "time" could mean "minute". Even if a clause, an essential fact or a statement has been encountered previously, rechecking is still required to confirm they are interpreted correctly in their specific contexts.

Lock the Program

The goal "lock the program" has the essential fact "the infusion rate is correctly set up", which requires the infusion rate calculation (involving VTBI) to be completed. This indicates that achieving "lock the program" is dependent on satisfying a condition. While "confirm the infusion rate" is an essential fact of goal "set up the infusion", this does not imply that achieving "lock the program" is dependent on "set up the infusion". Instead, the essential fact "activate the program lock" and the statement "$ProgramLocked = yes$" in the goal-related operation "ReadyToInfuse", of "set up the infusion", suggests a dependency in the opposite direction, implying "set up the infusion" may depend on "lock the program". On the other hand, "pause the infusion" is dependent on "starting the infusion" (hence dependent on the goal "set up the infusion"), which implies meeting all conditions for "set up the infusion".

Information in a Z specification is represented as attributes and properties rather than procedural dependencies, making it challenging to verify direct dependencies between goals. It may be the case that some other operations changed these properties and enabled certain goal-related operations, rather than implying dependency between two goals. What is certain is that, within this behavioural specification, this goal-related operation can not be enabled unless the essential facts of this goal are satisfied. The process, therefore, emphasizes conditions as reminders for developers to address certain requirements. Dependencies are further discussed in Chapter Five.

The essential facts "the syringe is correctly identified" in "load the syringe into the driver", and "the syringe is identified" in "set up the infusion", have the same meaning and represent the same condition. Using "correctly" illustrates how the writer may emphasize conditions using additional adverbs. While effective for emphasis, this writing style can sometimes introduce ambiguity, as developers cannot always be certain that similarly worded clauses share the same meaning. Therefore, scenarios across different contexts can not be assumed identical. Each scenario requires individual review to ensure an accurate interpretation of essential facts and goals. This reinforces the process's principle that context affects meaning and the nuances of conditions across scenarios.

Pause the Infusion

Similar to "set up the infusion", the state of infusion is represented in the matching statement "$SystemReady : YesNo$", thus "pause the infusion" is represented by setting property "SystemReady" to "no", which means the system is no longer ready to infuse, hence the infusion is paused.

Next, we demonstrate how to apply Step Three - analysing essential facts and forming predicates, to the infusion pump example.

4.1.3 Step Three Analysing Essential Facts and Forming Predicates

In Step Three, the Z specification with goal-related operations identified and the typed essential facts are used to form predicates for every essential fact in each goal. The requirements are interpreted and transformed into predicates, and the predicates are added to the respective predicate lists for each goal.

Load the Syringe Into the Driver

For goal "load the syringe into the driver", we must derive the meaning and form predicates for each of the essential facts shown in listing 43.

Listing 43 Assumption list for load the syringe into the driver

1. check battery is full
2. the syringe is correctly identified
3. confirm the syringe

Check Battery is Full

By reading the scenario and the narrative, the meaning of the essential fact is that in order to load the syringe into the driver, the battery must be full. The type of the essential fact is "YesNo", which is a boolean value that can represent whether the battery is full or not. To represent that the battery is full, the "Battery" should be "yes". In the goal-related operation "PreLoading" as shown in listing 31, the property related to "Battery" is $Battery' = Battery$, which means "Battery" remains unchanged. Therefore the predicate "$Battery = yes$" is formed, this is added to the predicate list for "Load the syringe into the driver".

The Syringe is Correctly Identified

By reading the scenario and the narrative, the meaning of the essential fact is that, the syringe brand should be identified, which also means it is not null. In the goal-related operation, Brand is changed to $brand?$ as given in $Brand' = brand?$. As there is no additional information for the correct brand, no brand can not be assumed to be correct, thus "$Brand \in SyringeBrand$" is formed for this essential fact, which means "Brand" is an element of the "SyringeBrand" type. This predicate is added to the predicate list for "Load the syringe into the driver".

Confirm the Syringe

The meaning of the essential fact is that the syringe must be confirmed to finish loading the syringe of the driver. The type of the essential fact is "YesNo", and the status of the syringe as suggested in the matching statement is "$SyringeOK : YesNo$". To represent that the syringe is checked, the value for "SyringeOK" from the available values "$yes|no$" should be "yes". Thus the predicate for the essential fact is "$SyringeOK = yes$", it is added to the predicate list for goal "load the syringe into the driver".

Step Three for goal "load the syringe into the driver" is now finished, the result is shown in 44. Next, goal "set up the infusion" is processed.

Listing 44 Predicate list for load the syringe into the driver

1. Battery = yes
2. $Brand \in SyringeBrand$
3. SyringeOK = yes

Set Up the Infusion

For goal "set up the infusion", we form predicates based on the meaning of conditions represented for each essential fact, as shown in listing 45.

Listing 45 Generalised assumption list for set up the infusion

1. the syringe is identified
2. confirm the syringe
3. activate the program lock
4. set the VTBI in millilitres
5. set the duration of infusion in hours and minutes
6. confirm the infusion rate

The Syringe is Identified

By reading the scenario and the narrative, the meaning of the essential fact is that, the syringe brand should be identified, which means it should be an element of the "SyringeBrand" type, given there is no null value in the type.

Thus the predicate formed is "$Brand \in SyringeBrand$", this predicate is added to the predicate list for "set up the infusion".

Confirm the Syringe

The meaning of the essential fact is that the syringe must be confirmed to finish setting up the infusion. The type of the essential fact is "YesNo", and the status of the syringe as suggested in the matching statement is "$SyringeOK : YesNo$". To represent that the syringe is checked, the value for "SyringeOK" from the available values "$yes|no$" should be "yes". Thus the predicate for the essential fact is "$SyringeOK = yes$", it is added to the predicate list for "set up the infusion".

Activate the Program Lock

The meaning of this essential fact is that the program lock must be activated to set up the infusion. The matching statement that represents the program lock, as identified earlier, is "$ProgramLocked : YesNo$". The available values of type "YesNo" are "yes | no". To represent that the program lock is activated, the value for "ProgramLocked" should be "yes". Thus "$ProgramLocked = yes$" is added to the predicate list.

Set the VTBI in Millilitres

The meaning of the essential fact is that the VTBI must have been set up to finish setting up the infusion. Based on the matching statements, VTBI is represented by "$VTBI : millilitres$", with type "millilitres", which is a natural number between zero and one hundred. When the VTBI is set, it should not be zero, nor a negative number, thus the VTBI should be greater than zero. The formed predicate is "$VTBI > 0$" (we do not include the range zero to one hundred here as it has already been stated elsewhere), it is also added to the predicate list.

Set the Duration of Infusion in Hours and Minutes

The meaning of this essential fact is that the duration of the infusion must have been set up before the infusion can be set up. That is, the duration of infusion is not zero. The duration of infusion is represented by "Hours" of type "hours" and "Minutes" of type "minutes", so they can not both be zero. Since the duration can not be negative, the predicate is "$Hours >= 0 \wedge Minutes >= 0 \wedge (\neg(Hours = Minutes = 0))$", this predicate is added to the predicate list.

Confirm the Infusion Rate

The essential fact implies that the infusion rate must be accurately calculated, yet no method for calculation is specified in the behavioural specification - it is merely implied. While practitioners are expected to verify the rate, no framework is provided to confirm calculation accuracy. While setting the rate may appear minor in behavioural specifications, from the evidence including our work and experience with the device, getting the rate right is crucial. However, as discussed [64], error often arises from user operation, not the pump itself, thus errors in rate calculations are frequently user-induced rather than mechanical.

It is assumed that the behavioural specification writer either knows how the rate is calculated or relies on others for this knowledge, yet this calculation is not explicitly documented. Similarly, it is presumed that pump users or developers understand the rate calculation and can verify its correctness. These assumptions are critical because what the user expects may not align with the device's actual function, yet there is no straightforward way to validate these assumptions, highlighting a gap in specification clarity.

The calculation for infusion rate should, ideally, be validated as part of the formal specification process. For our process, however, we establish a straight-

forward predicate to ensure the rate is set, implying it is non-zero. Since a negative rate is impossible, the predicate "$InfusionRate > 0$" is added to the list, representing that the infusion rate is initialised.

Step Three for goal "set up the infusion" is finished, and the result is shown in 46. Next, goal "lock the program" is processed.

Listing 46 Predicate list for set up the infusion

1. $Brand \in SyringeBrand$
2. SyringeOK = yes
3. ProgramLocked = yes
4. $VTBI > 0$
5. $Hours >= 0 \wedge Minutes >= 0 \wedge (\neg(Hours = Minutes = 0))$
6. $InfusionRate > 0$

Lock the Program

The meaning of "the infusion rate is correctly set up" is that to lock the program, the infusion rate must have been set up correctly. Similar to "set up the infusion", there is not enough information to verify the calculation is correct, thus a straightforward predicate to ensure the rate is set. The infusion rate is non-zero and not negative, thus "$InfusionRate > 0$" is added to the list, representing that the infusion rate is set.

Pause the Infusion

The meaning of "the infusion is running" is that, to pause the infusion, the infusion must be already running. The status of infusion is represented by the matching statement "$SystemReady : YesNo$", while the available values of "SystemReady" are "$yes|no$". To represent the system is already running, which means it is ready, we establish a predicate "$SystemReady = yes$", this predicate is added to the predicate list.

Step Three is completed, and the result is shown in listing 47. Next, we discuss observations and findings that emerged during this stage of the process.

Listing 47 Predicate list for T34 infusion pump example

1. Load the syringe into the driver
   1.1. Battery = yes
   1.2. $Brand \in SyringeBrand$
   1.3. SyringeOK = yes
2. Set up the infusion
   2.1. $Brand \in SyringeBrand$
   2.2. SyringeOK = yes
   2.3. ProgramLocked = yes
   2.4. $VTBI > 0$
   2.5. $Hours >= 0 \land Minutes >= 0 \land (\neg(Hours = Minutes = 0))$
   2.6. $InfusionRate > 0$
3. Lock the program
   3.1. InfusionRate > 0
4. Pause the infusion
   4.1. SystemReady = yes

Observations

For the essential fact "confirm the infusion rate", the main point is not that the rate should go unchecked but rather that the behavioural specification does not define the calculation explicitly. As discussed in the example, the actual calculation is not provided in the behavioural specification - only that the rate must exist and be greater than 0. It is assumed that the user understands the necessary rate calculation. While the behavioural specification can be consistent with the system (i.e. the Z specification), this does not necessarily mean completeness or that users/developers understand it fully. If the calculation steps in the behavioural specification are implied rather than explicitly outlined, there is potential for misinterpretation if these steps are translated into predicates. To avoid ambiguity, practitioners of the process deal with the information cautiously, ensuring they match user (stakeholder) understanding and system implementation (Z specification).

In practice, multiple scenarios are typically used to represent the same rule. For example, various input combinations for VTBI and infusion time yield different infusion rates, using the same calculation formula. Since behavioural specifications can not encompass every possible case, generalisation becomes necessary. Even with a high number of scenarios, coverage limitations inherent to model checking and unit testing remain. Generalisation allows the essential rule to apply across a broader range of possible values without manually defining each case.

The practitioner of the process can not be certain about the calculations simply from behavioural scenarios, even with several examples. The goal is not to confirm that the machine can perform the calculation - this is assured by the Z specification. However, the scenarios do suggest a narrative around these implied calculations. This aspect highlights an underlying assumption of the process: although the scenarios alone can not guarantee that the implementation of the system aligns with the user's assumption, the implementation is still implicitly believed to be correct.

In our approach, we combine information from the behavioural specification with a formal calculation formula identified in the Z specification. This formula adds certainty to the machine's correctness. However, our predicates only verify that the rate is "confirmed", without addressing the implicit human understanding required. Behavioural specification partially addresses the interaction between the system and users, while Z captures system behaviour. Nevertheless, a crucial element remains undocumented: the user's understanding and assumptions of the system. Although our work brings the behavioural specification and Z specification closer, it does not fully resolve this gap.

Features "set up the infusion" and "lock the program" both have "Bowen activates the program lock" clause, but they have different meanings. In "set up the infusion", "Bowen activates the program lock" is a requirement needed to enable goal-related operation "ReadyToInfuse", and produces the essential fact "activate the program lock", which translates to a predicate "$ProgramLocked = yes$". While in "lock the program", "Bowen activates the program lock" is the action of achieving the goal. This example demonstrates how the same phrase in a behavioural specification can carry different meanings depending on the context. While behavioural specifications effectively express system requirements, they also introduce risks of misinterpretation. Our process for comparing behavioural Z specifications helps unpack these complexities, offering a clearer insight into the requirements. A behavioural specification's seemingly simple scenario can contain substantial information. Specifically, clauses like "activate the program lock", may hold nuanced and varied meanings, even if they appear identical.

Next, we demonstrate how to apply Step Four - checking predicates, to the infusion pump example.

4.1.4 Step Four Checking Predicates

The predicates shown in listing 47 are now used to model-check their respective goal-related operations, against the Z specification.

Load the Syringe Into the Driver

The goal-related operation "PreLoading" should not be enabled if the list of predicates is not satisfied. In the predicate list, there are "$Battery = yes$", "$Brand \in SyringeBrand$" and "$SyringeOK = yes$". Thus there is a linear temporal logic formula to check in ProB:
$G(not((Battery = yes)\&(Brand \in SyringeBrand)\&(SyringeOK = yes))$

$=> not(e(PreLoading)))$

Using De Morgan's law, the formula is equivalent to:

$G(((not(Battery = yes))or(not(Brand \in SyringeBrand))or(not(SyringeOK = yes))) => not(e(PreLoading)))$

This means that globally (i.e. in every state) the operation "PreLoading" should not be enabled, if not satisfied: the value of Battery is "Yes", the value "Brand" is an element of the "SyringeBrand" type, and the value of SyringeOK status is "Yes". This check passed in ProB meaning the predicates are satisfied.

Set Up the Infusion

The goal-related operation "ReadyToInfuse" should not be enabled if the list of predicates is not satisfied. The formed linear temporal logic formula based on the predicates is checked:

$G(not((Brand \in SyringeBrand)\&(SyringeOK = yes)\&(ProgramLocked = yes)\&(VTBI > 0)\&((Hours >= 0)\&(Minutes >= 0)\&(not(Hours = Minutes = 0)))\&(InfusionRate > 0)) => not(e(ReadyToInfuse)))$

Using De Morgan's law, the formula is equivalent to:

$G(((not(Brand \in SyringeBrand))or(not(SyringeOK = yes))or(not(ProgramLocked = yes))or(not(Hours >= 0)\&(Minutes >= 0)\&(not(Hours = Minutes = 0)))or(not(InfusionRate > 0))) => not(e(ReadyToInfuse)))$

This means that globally (i.e. in every state) the operation "ReadyToInfuse" should not be enabled, if not satisfied: the value "Brand" is an element of the "SyringeBrand" type, the value of SyringeOK status is "Yes", the value of ProgramLocked status is "Yes", the value of 'VTBI' is greater than zero, the values of "Hours" and "Minutes" can not be negative, and they can not be zero at the same time, and the value of 'Infusion rate' is greater than zero. The check passed in ProB which means the predicates are satisfied.

Lock the Program

The goal-related operation "ProgramLock" should not be enabled if "$InfusionRate > 0$" is not satisfied, producing the following formula:
$G(not((InfusionRate > 0)) => not(e(ProgramLock)))$
This means globally, the operation "ProgramLock" should not be enabled if the value of "InfusionRate" is not greater than zero. The check passed in ProB which means the predicate is satisfied.

Pause the Infusion

The goal-related operation "PauseInfusion" should not be enabled if "$SystemReady = yes$" is not satisfied, producing the following formula:
$G(not((SystemReady = yes)) => not(e(PauseInfusion)))$
This means globally, the operation "PauseInfusion" should not be enabled if the value of "SystemReady" is not "yes". The check passed in ProB which means the predicate is satisfied.

Since the process was successfully conducted and the checks passed, we did not find any inconsistency between the behavioural and Z specifications.

4.2 Summary

We have demonstrated the application of the process to form and check predicates, to ensure consistency between the behavioural and Z specifications. This approach helps bridge the gap between the two specifications, providing confidence in their description of the requirements, though it does not fully guarantee correctness. The process goes beyond formalising behavioural specifications. Due to its use of natural language, behavioural specification contains nuanced information that the process can unpack, which is valuable for understanding the system requirements. In the next chapter, we will discuss the

findings from experiments as we explore the process of comparing behavioural and Z specifications.

Chapter 5

Discussion

In this chapter, we discuss the results of experiments and findings from exploring the process of comparing behavioural and Z specifications. We also discuss the trade-offs we made for our process and discuss further refinement that could be done after the process.

5.1 Negative Scenarios

The scenarios used in the process of comparing behavioural specifications and Z specifications described in Chapter Three are all positive scenarios. That is, the goal described in the narrative of that scenario is always achieved. For the vending machine example described in Step One in Chapter Three, as shown in listing 48, the goal is to "buy a snack with money", and the result is "vending machine dispenses a cookie", which means Bowen successfully bought a cookie. Now we look at a slightly different feature with negative outcomes in listing 49.

Listing 48 A vending machine scenario for buying a snack using money

```
Feature: Buying a snack using money
As a user
Bowen wants to buy a snack with money
So that Bowen can get a snack to eat

Scenario: Buying a cookie using money when the cookie
is in stock
Given cookie is in stock
When Bowen inserts 10 dollars
And Bowen selects cookie
Then the vending machine dispenses a cookie
```

Listing 49 Vending machine scenario with no cookie in stock

```
Feature: Buying a snack using money
As a user
Bowen wants to buy a snack with money
So that Bowen can get a snack to eat

Scenario: Buying a cookie using money when the cookie
is not in stock
Given cookie is not in stock
When Bowen inserts 10 dollars
And Bowen selects cookie
Then the vending machine does not dispense anything
```

The conditions are mostly the same, except "cookie is not in stock", and the result is "the vending machine does not dispense anything". If we apply the process described in Chapter Three to the scenario in listing 49, we will have essential facts "cookie is not in stock", "inserts 10 dollars" and "selects cookie". Now we obviously cannot apply Step Four in Chapter Three for this scenario. Because if "cookie is not in stock" is satisfied, which in the scenario it is, the outcome is a failure, which means the goal is not satisfied. Thus, the goal-related operation for "buy a snack with money" should not be enabled. We call scenarios with *unsuccessful* outcomes **negative** scenarios.

As discussed in section 3.2, in our process, we only consider *successful* scenarios where the goal of a feature is achieved in the outcomes, we call these **positive**

scenarios. At the start of Chapter Two, we explained that the behavioural specification necessarily describes the complete set of requirements for the system. The requirements discussed in the positive scenarios are a subset of all the requirements of the system under consideration, they are all mandatory but not necessarily complete. We can be sure that each condition (identified as essential facts) in a positive scenario has to be satisfied for the successful result (the goal being achieved) to be possible, but we can not be sure that the result is successful if each condition in a positive scenario is satisfied. That is, even if all the essential facts are satisfied, it is not guaranteed that the result is successful. But if any of the essential facts are not satisfied, it is guaranteed that the result is unsuccessful.

In the case of a *negative* scenario, while we understand that the combination of all conditions leads to an unsuccessful outcome, it remains unclear which condition, if not all, is responsible for the failure. Including negative scenarios in the process looks beneficial, as it can help identify conditions that may result in failure. However, their primary value is to remind stakeholders and developers to account for the edge cases, which can be used as a guide for programmers. Negative scenarios enhance the thoroughness of the development process, to give developers a better understanding of the edge cases that might otherwise be overlooked.

While negative scenarios may appear to be simple reversals of positive scenarios, where certain conditions are altered, they cannot be treated as such. Behavioural specifications are statements written in natural language, not formal logic. Therefore, we cannot reverse a condition in a scenario and guarantee it will produce the same effect in another scenario. Negative scenarios are implicitly covered by the positive scenarios - if any condition in a positive scenario

is not satisfied, the result is not successful, which is functionally equivalent to a negative scenario.

In [67], Lamport described a safety property as "something (bad) must not happen", and liveness property as "something (good) must happen". Although the positive scenarios describe the "good" result (goal achieved), they are not used to express *liveness* properties, as the conditions in a positive scenario are mandatory, but not necessarily sufficient. While the negative scenarios describe "bad" (goal failed to be achieved) results, they are not used to describe *safety* or *liveness* properties. We use the case of positive scenarios not being satisfied to cover safety properties, if any condition is not satisfied, something bad may happen.

As discussed previously, behavioural specifications do not have formal semantics. While we have provided rules for positive scenarios to infer the necessary conditions to successfully achieve the goal, they should implicitly cover the negative conditions that the negative scenarios can express. Comparing listing 48 and listing 49, the difference is whether "cookie is in stock". By interpreting this, we can deduce that "cookie is not in stock" is likely the condition causing the failure. However, when more than one condition varies, our confidence in pinpointing the exact cause of failure diminishes significantly. The flexibility offered by the natural language and structure used in behavioural specifications allows for great expressiveness, but imposing too many rules or rigid interpretations risks producing an artificial and less applicable result. While using negative scenarios to check properties holds potential for further exploration, the amount of rules required to achieve sufficient confidence is unknown, it lies beyond the scope of this thesis and is considered future work.

5.2 Sequence and Ordering of Events

The process of comparing behavioural and Z specifications given in Chapter Three described how information in the scenarios is extracted as essential facts. The essential facts are extracted and listed independently - one essential fact is not dependent on another, and every requirement described in an essential fact can be satisfied separately, without considering whether one requirement needs to be satisfied to enable another. By treating the essential facts independently, we can consider the requirements individually or jointly for certain goals. This benefits us when we want to check if a constraint is reflected in the Z specification. However, such an approach does not take the relationship between the essential facts into account, it does not consider requirements that are dependent on one another - a requirement can be satisfied only if another requirement is satisfied. This dependency relationship between the two requirements is not captured in our process. Next, we discuss the meaning of dependency and the transitivity of dependency, then we discuss what dependency means for ordering, and explain why ordering is not considered in our process.

Dependency

Between two essential facts EA and EB, where EA is dependent on EB, EB must be satisfied before EA can be satisfied. In first-order logic, predicate PA (from EA) is dependent on predicate PB (from EB), the truth of PA relies on the truth of PB. In logical terms, this is expressed as **PB implies PA** (denoted as $PB \Rightarrow PA$).

Consider a feature for entering passwords as shown in listing 50:

Listing 50 A banking system scenario for logging in using password

```
Feature: Log into the bank system using password
As a user
Bowen wants to log into the bank system
So that Bowen can manage his money

Scenario: Successfully logging into the bank system
with the correct password
Given Bowen opens the log-in menu
When Bowen types in the correct password
And Bowen presses the log-in button
Then Bowen successfully logged into the bank system
```

By applying the process discussed in Chapter Three, the goal "log into the bank system" has the following essential facts in the assumption list: "open the log-in menu", "types in the correct password", "presses the log-in button", so these essential facts must be satisfied before a user can successfully log into the bank system. Realistically in the banking system, a user cannot type in the password unless the log-in menu is opened, thus the essential fact "types in the correct password" is in fact dependent on "open the log-in menu". Similarly, when no password is typed in (no matter whether the password is correct or not), pressing the log-in button either triggers a pop-up window that states "Password can not be empty", or the system simply does not respond to the button press. Therefore, to *successfully* achieve the goal "log into the bank system", the essential fact "presses the log-in button" is dependent on "types in the correct password".

Transitivity

For essential facts, we define **transitivity**: given three essential facts A, B, and C, A is dependent on B, B is dependent on C, then A is also dependent on C. This means the dependency relationship is **transitive**, the predicate PA formed from A is also dependent on the predicate PC formed from C. For the example shown in listing 50, "presses the log-in button" is dependent on "types

in the correct password", "types in the correct password" is dependent on "open the log-in menu". Therefore, "presses the log-in button" is also dependent on "open the log-in menu". Since A is dependent on B, which is dependent on C, transitivity ensures that, if essential fact A is satisfied, B must also be satisfied since A is dependent on it, and C must also be satisfied since B is dependent on it, this information may be useful in model checking the properties.

Ordering and Suggested Meaning in Behavioural Specification

When the *transitive dependency* relationships exist between the essential facts, they have to be satisfied in a certain order, so an essential fact A can be "enabled" for the next essential fact B that is dependent on A. However, such "ordering" is usually not explicitly stated in the behavioural specification. In the original article [76] where Dan North introduced behavioural specification, the scenario structure as described in listing 51, suggests some form of dependency. We can infer that the "event" in the "When" clause is dependent on the "context" provided in the "Given" clause, and the "outcome" in the "Then" clause is dependent on the "event" in the "When" clause (hence dependent on "Given" through transitivity). This dependency relationship is implied by the natural language terms "Given" and "When", but it remains weak, as it is not explicitly stated. Therefore, we can not definitively determine whether the events described in the "When" clause can only occur under the specified "Given" context.

Listing 51 Dan North scenario draft

```
Given some initial context (the givens),
When an event occurs,
Then ensure some outcomes.
```

In a more concrete example from [76], North described a withdraw cash scenario, as shown in listing 52. While the "When" clause may suggest that "cus-

tomer requests cash" is dependent on "card is valid" condition, the relationship between "customer requests cash" and "dispenser contains cash", and the relationship between "dispenser contains cash" and "card is valid", remains unclear. After examining the scenarios written by North and other BDD scenarios, we conclude that there is no formal enforcement of a dependency relationship between the "Given" and "When" clauses. These dependencies are often left implicit, allowing for ambiguity in how different conditions relate to one another.

Listing 52 Dan North withdraw cash scenario

```
Feature: Customer withdraws cash
As a customer,
I want to withdraw cash from an ATM,
so that I don't have to wait in line at the bank.
Scenario: Account is in credit

Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned
```

Since we are unable to infer dependency information from the behavioural specifications, it is unsafe to assume that a dependency relationship exists between the 'Given' and 'When' clauses, or even within those clauses themselves. The only certainty we have is that the conditions represented in the "Given" and the "When" clauses combined, will always imply the outcome in the "Then" clause. Thus the only confirmed dependency relationship, is that the outcome in the "Then" clause is dependent on both the context in the "Given" clause and the event in the "When" clause. We captured this information in the process described in Chapter Three, that the contexts in the "Given" clause and the events in the "When" clause are identified as essential

facts, and the outcome in the "Then" clause is identified as the goal-related operation being enabled. As we are focusing solely on positive scenarios, the outcomes are always successful, and the goal-related operation being enabled is always dependent on the essential facts being satisfied. As Section 3.1.4 in Chapter Three described, globally, if not all essential facts are satisfied, the goal-related operation should not be enabled.

While introducing the dependency relationship can impose additional constraints on the requirements to achieve the goal, enriching the information carried in the behavioural specification, the level of detail in the behavioural specifications is too abstract to accurately infer such relationships. Assuming the dependency relationship without clear evidence risks weakening the correctness of the process, rather than supporting it. Therefore, after careful consideration, we have opted not to include dependency relationships in our process, we treat the essential facts as independent and they can be satisfied independently.

5.3 Refinement

After applying our process described in Chapter Three, we have formalised the conditions described in the behavioural specification and checked for inconsistency between the behavioural and Z specification. The next logical step is to explore a concrete application of the transformed predicates. Refinement [55] is a technique often used for transforming an abstract specification into a more concrete one, ensuring that the more concrete version preserves the properties and correctness of the original specification. Refinement can also serve as a comparison tool between two artefacts that describe the same system at different levels of abstraction. In this sense, the predicates formed in our process can be viewed as a refinement of the behavioural specification.

In formal methods, a formal specification can be refined by gradually adding more details, which changes the formal specification into something more concrete, which decreases the level of abstraction of the formal specification. Another refinement approach is removing unwanted, non-deterministic elements from the existing specifications, which gradually decreases the abstraction level of the specifications. Refinement can be applied to formal specifications, to extract relations or elements that describe the core requirements of the desired systems. With such relations, we can then check if there is an equal or subset relationship between them. Note that in our research, after applying the process of comparing behavioural and Z specifications, only the Z specification can be refined, the behavioural specifications will remain unchanged.

After checking the behavioural specification and Z specification are consistent, refinement can be done on the Z specification. As a system is being implemented, the formal specification of the system is also being refined. Two types of refinement are discussed in this work that can be applied to a Z specification, the first is data refinement and the second is operation refinement. Doing refinement may require adjusting the predicates which impacts the consistency, thus we need to get back to the goal-related operations and matching statements in the Z specification and decide whether we need to adjust the predicates, also whether the consistency has been preserved.

Suppose the original Z specification FA is refined into FA', FA' would be a subset of FA, the refinement often involves simplifying the specification. We now discuss how refinements can impact the consistency between the behavioural specification and the Z specification.

5.3.1 Data Refinement

Data refinement refers to the process of replacing a data type, data range, or structures with more concrete implementations while preserving correctness [94]. Suppose a data type in the abstract specification, denoted as A, is refined to a corresponding data type in the refined specification, denoted as A'. A and A' share the same indexing set. A retrieve relation R defines how data in the abstract type A can be retrieved from, or related to, the more concrete data in A'. This relation must be formally proved, ensuring that the refinement maintains the behaviour of the system.

Furthermore, the retrieve relation between FA and FA' provides the means to transition between the abstract and concrete specifications. It is important to note that, the original set of predicates P derived from Step Four of our process, was used to demonstrate consistency between the behavioural specification B and the abstract specification FA. A question is whether P can still be used to show consistency between B and FA', and if not, whether P can be refined to P' that can show consistency between B and the refined Z specification FA'.

After applying data refinement, a retrieve relation should already exist based on the decisions made during the refinement process, providing us with a direct schema to follow. Now we explore how we can refine a data type. We can obtain a simple definition of refinement if we restrict our attention to total relations. If R and S are total relations, then R refines S exactly when R $\subseteq$ S. In cases where S relates the same element x to two distinct elements y_1 and y_2, R may resolve this ambiguity by omitting either the pair (x, y_1) or (x, y_2). Therefore, the retrieve relation is not simple in this case. Now we consider how we perform the refinement.

Using the T34 example discussed in Chapter Four, the state schema has a property "Battery : YesNo", which indicates whether the battery is full, its data type is YesNo, as shown in listing 53, "yes" indicates the battery is full, and "no" indicates the battery is not full.

Listing 53 Data type YesNo

$$YesNo ::= yes|no$$

Now we consider refining this to a more concrete data type "Level", as shown in listing 54, where the battery level is "low", "medium" or "high".

Listing 54 Data type Level

$$Level ::= low|medium|high$$

"BatteryYesNo" and "BatteryLevel" are concrete data types that appear to use different indexing sets. However, it may be possible to transform "BatteryYesNo" or "BatteryLevel" so they share the same indexing set. For instance, we can map "No" in "BatteryYesNo" to "Low" in "BatteryLevel", indicating that the battery is insufficient for operation. But mapping "Yes" in "BatteryYesNo" is more complex, as it encompasses both "Medium" and "High" in "BatteryLevel". We can express "Yes" as "Medium or High", making the data represented in BatteryLevel a subset of the data represented in BatteryYesNo.

To demonstrate that a retrieve relation exists between the original specification and the refined specification, the following schema replaces BatteryYesNo with BatteryLevel:

Listing 55 Retrieve relation from BatteryLevel to BatteryYesNo

$$
\begin{array}{|l}
\hline
RetrieveBattery \\
BatteryLevel \\
BatteryYesNo \\
\hline
BatteryYesNo = Yes \longleftrightarrow \\
(BatteryLevel = High \lor BatteryLevel = Medium) \\
BatteryYesNo = No \longleftrightarrow BatteryLevel = Low \\
\hline
\end{array}
$$

Certain requirements must be met for the retrieve relation shown in listing 55 to be considered a *forward simulation*[70]. A forward simulation demonstrates transitions from an abstract specification to a more concrete specification. The detailed calculations or formal proof for forward simulation are beyond the scope of this work and will not be discussed here. The refinement process involves replacing the binary "Yes/No" data type in BatteryYesNo with the ternary "Low/Medium/High" data type in BatteryLevel. This transition is not a trivial computation. Manual changes to the variables in the formal specification are required, specifically deciding whether to substitute Yes with "Medium" or "High". This ambiguity requires a decision to be made by the developer based on the system's intended behaviour. However, the mapping for "No" will always correspond to "Low" unambiguously in the refined specification.

While we can substitute "Battery = Yes" with either "BatteryLevel = High" or "BatteryLevel = Medium" in the formal specification FA, both cases can be represented within a single predicate. For example, in the predicate PA:
$G(not(SyringeOK = yes \& Battery = Yes) => not(e(PreLoading)))$
We can refine it to PA' :
$G(not(SyringeOK = yes \& (Battery = Medium \lor Battery = High) =>$
$not(e(PreLoading)))$

This allows us to account for both refined conditions in the same predicate.

Reflecting on how this refinement affects the predicates generated during the process, applying the retrieve relation directly to substitute the refined data types is insufficient, as it does not fully capture how the refinement impacts the behaviour and data in relation to the behavioural specification. As a result, the matching statements for the essential facts in the predicates must be reviewed. If these statements are altered by the refinement, the corresponding predicates must also be reconsidered to determine if further refinement is necessary.

5.3.2 Operation Refinement

Another type of refinement is operation refinement. Operation refinement refers to the process of refinement specific operation without reference to the other operations [41]. There are various ways of conducting operation refinement, including completing a relation, directly weakening preconditions and strengthening post-conditions, and sets of reasonable deterministic implementations.

Although we did not perform a comprehensive experiment on refining Z specification operations, we identified a few critical points that should be considered. After the refinement, the goal-related operations need to be reviewed, as these operations reflect the behaviour described in the behavioural specification. If any of these operations are refined, this indicates the behaviour may have been altered, thus the corresponding predicate list for the goal achieved by that goal-related operation must also be reviewed and refined where necessary. If any predicates require refinement, the changes (refinement) made to the Z spec-

ification should be reflected in the predicates, while preserving their original meaning, so the behaviour of that goal-related operation remains consistent. This ensures that Step Four of our process can still be performed, to check that the consistency between the behavioural and Z specifications is preserved.

5.3.3 Refinement in Our Process

Note that in our process of comparing the behavioural specification and Z specification, applying generalisation to an essential fact does not alter the meaning of the essential fact, as the generalisation was based on the original intent of the writer. This can be viewed as a form of data refinement, where the value represented by the essential fact is refined to a range of values. Since the concrete essential facts are generalised to their types, this does not affect our data types in the Z specification, as they are unchanged. Likewise, goal-related operations are unaffected, as the goals themselves are not changed through generalisation. It is important to emphasise that we are not modifying the behavioural specifications, we are only refining the Z specification and assessing how the refinement impacts the predicates. Therefore, no new constraints are introduced from the perspective of the behavioural specification.

5.4 Summary

In this chapter, we discussed several aspects of the behavioural specification which are not included in our process, including negative scenarios, dependency, sequence, and orderings. We have also discussed the refinement of the Z specification, providing a brief example of refining the data type used for the battery.

Chapter 6

Conclusions

6.1 Overview

In the introduction of this thesis, we hypothesised that behavioural-driven development and formal methods can be used together to describe safety-critical interactive systems to support correctness. In this research, we presented a complementary approach to integrate behavioural and Z specifications and check for consistency. Conditions described in the behavioural specifications are formalised as first-order logic predicates, and compared with their corresponding statements in the Z specification. This process reveals inconsistencies between the two artefacts and provides a better understanding of the system requirements. Examples of applying our process have been given and they have shown the benefits of using this process.

To prove the hypothesis, we addressed the following research questions:

1. What are the challenges in comparing behavioural specifications with formal specifications?

2. How do we address the challenges identified in research question one to

make behavioural specifications comparable with formal specifications?

3. How do we use the comparison in an integrated approach to support safety-critical interactive system development?

We addressed Research Question One in Chapter Two, highlighting that behavioural and formal specifications can not be compared directly due to different syntax, semantics, and levels of abstraction. Chapters Four and Five, further explored these distinctions, discussing why direct integration is unfeasible and the challenges we encountered when we attempted to compare the two artefacts.

In Chapter Two we discussed how informal artefacts can be formalised to be comparable with formal artefacts. In Chapter Three, we discussed the problems that need to be solved to enable the comparison and the methodology used to solve these problems. We then described the process of comparing the two specifications, including how to find the conditions described in the behavioural specification, and use the information in the Z specification to support a better understanding of these conditions. These conditions are further formalised and checked against the Z specification. These discussions address Research Question Two.

In Chapter Four, we demonstrated how to use the proposed approach to check consistency between the behavioural and Z specifications for the Niki T34 infusion pump. We also demonstrated how the process can unpack the hidden information in the seemingly simple behavioural specification, which provides a deeper insight into the system requirements of the safety-critical interactive system. This addresses research Question Three.

6.2 Contributions

The contributions of this thesis are as follows. We have explored the relationship between the structural behavioural specifications and Z specifications, in which the design focus, syntactic structure, and semantic representation differences between these artefacts are addressed. Our investigation demonstrates that, while the two artefacts are often tailored and used for different aspects of a system, they are complementary and deeply interconnected. The formal specification helps mitigate ambiguities inherent in the natural language of behavioural specifications, offering a precise view to interpret the conditions. Conversely, the behavioural specification provides a bridge for involving the business stakeholders, imparting domain-specific knowledge that is often overlooked by the developers.

Our research led to a systematic approach to comparing the two artefacts. In the developed approach, the essential facts in the behavioural specification are understood and formalised, which is checked in the ProB model checker against the Z specification, to ensure consistency between the two specifications. While the behavioural specification is easy to use, it often contains implied or ambiguous information. Our approach provides a structured way to unpack these ambiguities.

The developed approach was applied to the Niki T34 infusion pump, which is a safety-critical interactive system. In addition to checking the consistency between the behavioural and Z specifications, the case study demonstrated how our process can be used to unpack the hidden information and potential inconsistencies within the behavioural specification. By integrating both behavioural and Z specifications, the understanding of the system requirements is

enhanced, which supports the correctness of the safety-critical interactive system under consideration. This process also brings the business stakeholders and the developers closer, by bridging the communication gap created by different perspectives between them. Our work provides a foundation for further research into integrating behavioural and formal methods, which emphasises the need for a larger coverage of requirements and resolving ambiguities that lie within the specifications.

6.3 Limitations and Future Work

While the process we developed can be applied to any behavioural specification, a matching Z specification that specifies the same system is required. Several aspects of the behavioural specification such as negative scenarios and dependencies between goals or essential facts, are deliberately excluded. These elements are commonly used in behavioural specifications, and may contain valuable information including boundary conditions, edge cases and error handling. Incorporating these aspects into our process may capture a broader range of system requirements, unravel more hidden information that is overlooked, and identify inconsistencies that may arise in the edge cases.

Since the process has been tested on limited examples, due to the limited availability of real-world Z and behavioural specifications, a key direction for future work is to conduct a longitudinal studying involving a real-world software development team to evaluate and refine the proposed process. Integrating the process into a real-world ongoing system development lifecycle would offer critical insights into the evaluation of the process for its effectiveness, usability and adaptability. Moreover, a larger example would help in refining the process and also uncover overlooked aspects, it can also help observe if the edge/boundary conditions discussed in Chapter Five need to be handled in the process or not.

These aspects are difficult to fully capture in smaller examples, but they are often critical in real-world large systems. However, such a study would require a significant time and resources because multiple development teams and cycles are required for the iterative feedback and refinement. Despite this, the outcome could potentially be a more robust and generalised method that can better support industry-level specifications for improving system correctness.

The current method for forming predicates works well for straightforward relationships which are intended for behavioural specifications, but this method may struggle with representing more complex relationships or conditions. A technique that copes well with such complicated information can support the correctness of safety-critical interactive system development, and enhance the accuracy of formalising the information in the behavioural specification. The current notion of consistency used in the process could also be expanded and refined to cover more aspects and support a higher level of correctness, such as real-time properties, industrial standards or liveness properties.

Furthermore, the work can be extended to include other formal notations, such as B-method or Alloy that were introduced in Chapter Two, which could make the process more versatile and applicable to a wider range of systems. Incorporating these notations can benefit from their unique strengths, such as Alloy's lightweight modelling capabilities and B-method's refinement theories, which complement Z notation. This flexibility could also enable the developers to choose the method that is tailored to their specific needs for easier access and better coverage of system requirements.

While our approach is fully manual, automation may prove useful for certain steps of the process. For example, finding essential facts and checking

consistencies, could learn from the automated translations between Z and B-method. Developing such automation may also inspire mechanisms to address ambiguities from the natural language used in the behavioural specification. Techniques such as natural language processing (NLP) or machine learning could potentially assist in this automation, making the process more scalable.

As discussed in previous chapters, refinement is a core aspect of formal methods, which is used to transform an abstract specification into a more concrete one. This thesis experimented with refinement but did not go into depth or come to a solution. However, the following questions arise from this topic:

1. What refinement theory can we apply to the artefacts?
2. Can we show that the consistency between the behavioural specification and the refined formal specification is retained after the refinement?
3. If consistency changes after refinement, what does it mean? Does it indicate an error in the refinement process? Does it indicate that the behavioural specification contains insufficient detail to match the refined system?
4. How do we make use of the information?

The matching statements concept in our work helps track the statements that are relevant to the essential facts of the behavioural specification, which may be changed after refinement. In addition, the predicates formed in our process could be used as a basis for checking the consistency after refinement, with a proper retrieve relation established. As refinements often reveal insights into the system which may not appear obvious in the original specification, this could alert the stakeholders to review the behavioural specifications that ambiguity lies within the behavioural specification and needs to be resolved.

6.4 Conclusion

Using the behavioural and formal specifications independently has its unique strengths and weaknesses, but safety-critical interactive system development requires more than each approach can offer individually. When the two artefacts are used together issues may arise, due to the gap between semantic and design focus. This research presents a systematic approach for integrating the behavioural and formal specifications by formalising conditions described in the behavioural specification and checking them against their corresponding parts in Z specifications. Inconsistency is revealed should any part of the process fail, indicating the specifications require revision.

Furthermore, this research emphasises that the behavioural and formal specifications are complementary. While this integration does not guarantee correctness, it can effectively improve the correctness of safety-critical interactive system development, by making sure that the conditions in the behavioural specification are satisfied in the Z specification. In addition, this process bridges the communication gap between the business stakeholders and developers by enhancing their collaboration. Our work also addresses challenges in integrating these specifications, stemming from their different syntax, semantics, levels of abstraction and design focus. Addressing these challenges lays a foundation for further exploration into this integration between informal and formal artefacts.

This work can be fully applied in its complete form to identify inconsistencies between behavioural and Z specifications for a safety-critical interactive system, unpacking ambiguities due to the use of natural language. The approach can unravel information and requirements hidden in the behavioural specification, which is particularly useful for the developers to improve the

correctness of the system. In summary, this work successfully solves the problem we identified in Chapter One, and offers more benefits beyond its initial scope.

Appendices

Appendix A

NiKi T34 Infusion Pump Z Specification

$YesNo ::= yes | no$

$SyringeBrand ::= BDPlastipak | Terumo$

$millilitres : \mathbb{P}\,\mathbb{N}$

$millimetres : \mathbb{P}\,\mathbb{N}$

$hours : \mathbb{P}\,\mathbb{N}$

$minutes : \mathbb{P}\,\mathbb{N}$

$millilitresperhour : \mathbb{P}\,\mathbb{N}$

$brands : SyringeBrand \leftrightarrow SyringeBrand$

$millilitres = 0 \,..\, 100$

$millimetres = 0 \,..\, 10$

$hours = 0 \,..\, 24$

$minutes = 0 \,..\, 59$

$millilitresperhour = 0 \,..\, 100$

$brands = \{(Terumo, BDPlastipak)\}$

$T34$

$Battery : YesNo$

$KeyPadLocked : YesNo$

$ProgramLocked : YesNo$

$TechMenuLocked : YesNo$

$Brand : SyringeBrand$

$SyringeSize : millilitres$

$VolumeRemaining : millilitres$

$PlungerPosition : millilitres$

$SyringeOK : YesNo$

$BarrelOK, CollarOK, PlungerOK : YesNo$

$SystemReady : YesNo$

$VTBI : millilitres$

$Hours : hours$

$Minutes : minutes$

$InfusionRate : millilitresperhour$

Init

$T34$

$Battery = yes$

$KeyPadLocked = no$

$ProgramLocked = no$

$TechMenuLocked = yes$

$Brand = Terumo$

$SyringeSize = 1$

$VolumeRemaining = 0$

$PlungerPosition = 1$

$SyringeOK = no$

$BarrelOK = CollarOK = PlungerOK = no$

$SystemReady = no$

$VTBI = 1$

$Hours = 1$

$Minutes = 1$

$InfusionRate = 0$

KeyPadLock

$\Delta T34$

$SyringeOK = yes$

$BarrelOK = CollarOK = PlungerOK = yes$

$Battery' = Battery$

$KeyPadLocked = no \Rightarrow KeyPadLocked' = yes$

$KeyPadLocked = yes \Rightarrow KeyPadLocked' = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VTBI'$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

ProgramLock

$\Delta T34$

$SyringeOK = yes$

$BarrelOK = CollarOK = PlungerOK = yes$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no \Rightarrow ProgramLocked' = yes$

$ProgramLocked = yes \Rightarrow ProgramLocked' = no$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VTBI'$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

$VTBI \leq SyringeSize$

$Hours > 0 \lor Minutes > 0$

$VTBI > 0$

$PlungerPosition > 0$

$VolumeRemaining > 0$

$InfusionRate > 0$

TechMenuLock

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked = no \Rightarrow TechMenuLocked' = yes$

$TechMenuLocked = yes \Rightarrow TechMenuLocked' = no$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VTBI'$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

IncreaseVTBI

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$PlungerOK' = PlungerOK$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$SystemReady' = SystemReady$

$VTBI < 100$

$VTBI' = VTBI + 1$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

DecreaseVTBI

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI > 0$

$VTBI' = VTBI - 1$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

SetVolumeRemaining

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VTBI'$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SyringeOK = yes$

$BarrelOK = CollarOK = PlungerOK = yes$

$SystemReady' = SystemReady$

$VTBI > 1$

$VTBI \leq SyringeSize$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

IncreaseDuration

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$(Minutes < 59 \land Hours < 24) \Rightarrow (Minutes' = Minutes + 1 \land Hours' = Hours)$

$(Minutes = 59 \land Hours < 24) \Rightarrow (Minutes' = 0 \land Hours' = Hours + 1)$

$(Minutes = 0 \land Hours = 24) \Rightarrow (Hours' = Hours \land Minutes' = Minutes)$

$InfusionRate' = InfusionRate$

DecreaseDuration

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$(Minutes = 0 \land Hours = 0) \Rightarrow (Hours' = Hours \land Minutes' = Minutes)$

$(Minutes > 0) \Rightarrow (Minutes' = Minutes - 1 \land Hours' = Hours)$

$(Minutes = 0 \land Hours > 0) \Rightarrow (Minutes' = 59 \land Hours' = Hours - 1)$

$InfusionRate' = InfusionRate$

CalculateRate

$\Delta T34$

$SyringeOK = yes$

$BarrelOK = CollarOK = PlungerOK = yes$

$SystemReady = no$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$Minutes > 0 \lor Hours > 0$

$InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes)$

ReadyToInfuse

$\Delta T34$

$SyringeOK = yes$

$BarrelOK = CollarOK = PlungerOK = yes$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = yes$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady = no$

$SystemReady' = yes$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

SelectBrand

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

PlungerForward

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition < 100$

$PlungerPosition' = PlungerPosition + 1$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK = no$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

PlungerBack

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition > 0$

$PlungerPosition' = PlungerPosition - 1$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK = no$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

SetPlunger

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerPosition \geq SyringeSize$

$PlungerOK = no \Rightarrow PlungerOK' = yes$

$PlungerOK = yes \Rightarrow PlungerOK' = no$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

SetBarrel

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK = no \Rightarrow BarrelOK' = yes$

$BarrelOK = yes \Rightarrow BarrelOK' = no$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

SetCollar

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK = no \Rightarrow CollarOK' = yes$

$CollarOK = yes \Rightarrow CollarOK' = no$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

IncreaseSyringeSize

$\Delta T34$

$SyringeOK = no$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize < 100$

$SyringeSize' = SyringeSize + 1$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

DecreaseSyringeSize

$\Delta T34$

$SyringeOK = no$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize > 0$

$SyringeSize' = SyringeSize - 1$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

SetSyringeSize

$\Delta T34$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = no$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand$

$SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeSize \leq PlungerPosition$

$SyringeOK = no \Rightarrow SyringeOK' = yes$

$SyringeOK = yes \Rightarrow SyringeOK' = no$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = Hours$

$Minutes' = Minutes$

$InfusionRate' = InfusionRate$

PreLoading

$\Delta T34$

$brand? : SyringeBrand$

$syringeSize? : millilitres$

$volumeRemaining? : millilitres$

$plungerPosition? : millilitres$

$VTBI? : millilitres$

$SyringeOK = no$

$BarrelOK = CollarOK = PlungerOK = no$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = brand?$

$SyringeSize' = syringeSize?$

$VolumeRemaining' = volumeRemaining?$

$PlungerPosition' = plungerPosition?$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK$

$CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady' = SystemReady$

$VTBI' = VTBI?$

$Hours' = 24$

$Minutes' = 0$

$Minutes > 0 \lor Hours > 0$

$InfusionRate' = (60 * VTBI) \div ((60 * Hours) + Minutes)$

Tick

$\Delta T34$

$battery? : YesNo$

$syringeOK? : YesNo$

$barrelOK? : YesNo$

$collarOK? : YesNo$

$plungerOK? : YesNo$

$step? : millimetres$

$Hours > 0 \lor Minutes > 0$

$Battery' = battery?$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = yes$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand \land SyringeSize' = SyringeSize$

$VolumeRemaining' = ((10000 * VolumeRemaining)$

$-(10000 * InfusionRate \div 3600)) \div 10000$

$PlungerPosition' = PlungerPosition - step?$

$SyringeSize \leq PlungerPosition$

$SyringeOK' = syringeOK?$

$BarrelOK' = barrelOK?$

$CollarOK' = collarOK?$

$PlungerOK' = plungerOK?$

$SystemReady = yes \land SystemReady' = SystemReady$

$VTBI' = VTBI$

$Hours' = (((Hours * 3600) + (Minutes * 60)) - 1) \div 3600$

$Minutes' = ((((Hours * 3600) + (Minutes * 60)) - 1) \mod 3600) \div 60$

$InfusionRate' = InfusionRate$

PauseInfusion

$\Delta T34$

$SyringeOK = yes$

$BarrelOK = CollarOK = PlungerOK = yes$

$Battery' = Battery$

$KeyPadLocked' = KeyPadLocked$

$ProgramLocked = yes$

$ProgramLocked' = ProgramLocked$

$TechMenuLocked' = TechMenuLocked$

$Brand' = Brand \land SyringeSize' = SyringeSize$

$VolumeRemaining' = VolumeRemaining$

$PlungerPosition' = PlungerPosition$

$SyringeOK' = SyringeOK$

$BarrelOK' = BarrelOK \land CollarOK' = CollarOK$

$PlungerOK' = PlungerOK$

$SystemReady = yes \land SystemReady' = no$

$VTBI' = VTBI$

$Hours' = Hours \land Minutes' = Minutes$

$InfusionRate' = InfusionRate$

ScrollInfoListDown

$\Xi T34$

ScrollInfoListUp

$\Xi T34$

Bibliography

[1] Alloy tutorial. https://alloytools.org/tutorials/online/, Last accessed 10 December 2024.

[2] Atelier B. https://www.atelierb.eu/en/, Last accessed 10 December 2024.

[3] Balsamiq: Wireframe with ease. https://balsamiq.com, Last accessed 10 December 2024.

[4] Context. https://corporate.postoffice.co.uk/en/horizon-scandal-pages/context, Last accessed 10 December 2024.

[5] Expert it witness outsmarted an 'aggressive' post office to get to truth after inspection 'madness'. https://www.computerweekly.com/news/366567874/Expert-IT-witness-outsmarted-an-aggressive-Post-Office-to-get-to-truth-after-inspection-madness, Last accessed 10 December 2024.

[6] Gherkin reference. https://cucumber.io/docs/gherkin/reference/, Last accessed 10 December 2024.

[7] Overture tool. https://www.overturetool.org/

publications/books/vdmtools.html, Last accessed 10 December 2024.

[8] ProB. https://prob.hhu.de/, Last accessed 10 December 2024.

[9] ProZ. https://prob.hhu.de/w/index.php?title=ProZ, Last accessed 10 December 2024.

[10] Pvsio-web: A javascript library to connect to and communicate with a Pvsio process using websockets. https://github.com/pvsioweb/pvsio-web, Last accessed 10 December 2024.

[11] Software testing - bug vs defect vs error vs fault vs failure. https://www.geeksforgeeks.org/software-testing-bug-vs-defect-vs-error-vs-fault-vs-failure/, Last accessed 10 December 2024.

[12] VDM tools. https://vdmtools.docs.cern.ch/, Last accessed 10 December 2024.

[13] What is Cucumber? https://cucumber.io/docs/guides/overview/, Last accessed 10 December 2024.

[14] Z/EVES eclipse prover IDE. https://czt.sourceforge.net/eclipse/zeves/, Last accessed 10 December 2024.

[15] NIKI T34 operating guidelines, 2011. https://www.wnswphn.org.au/uploads/documents/ePAF/20b%20-%20WNSW%20LHD%20NIKI%20T34%20Syringe%20Driver%20Operating%20Guidelines%20(PD2011_106).pdf, Last

accessed 10 December 2024.

[16] J.-R. Abrial. *The B-book: Assigning programs to meanings*. Cambridge University Press, 1996.

[17] S. Aujla, T. Bryant, and L. Semmens. A rigorous review technique: Using formal notations within conventional development methods. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 247–255, 1993.

[18] A. Bacon. *A philosophical introduction to higher-order logics*. Routledge, 2023.

[19] K. Beck. *Test driven development: By example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[20] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, M. Fowler W. Cunningham, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile software development, 2001. `https://agilemanifesto.org/`, Last accessed 10 December 2024.

[21] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, M. Fowler W. Cunningham, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Principles behind the Agile manifesto, 2001. `https://agilemanifesto.org/principles.html`, Last accessed 10 December 2024.

[22] P. Beynon-Davies. Human error and information systems failure: The case of the london ambulance service computer-aided despatch system project. *Interact. Comput.*, 11(6):699–720, 1999.

[23] L. P. Binamungu, S. M. Embury, and N. Konstantinou. Characterising the quality of behaviour driven development specifications. In V. Stray, R. Hoda, M. Paasivaara, and P. Kruchten, editors, *Agile processes in software engineering and extreme programming - 21st International Conference on Agile Software Development, XP 2020, Copenhagen, Denmark, June 8-12, 2020, Proceedings*, volume 383 of *Lecture Notes in Business Information Processing*, pages 87–102. Springer, 2020.

[24] J. Bowen and M. Hinchey. Ten commandments of formal methods ...Ten years later. *Computer*, 39:40 – 48, 02 2006.

[25] J. Bowen and S. Reeves. Formal models for user interface design artefacts. *ISSE*, 4(2):125–141, 2008.

[26] J. Bowen, B. Weyers, and B. Liu. Creating formal models from informal design artefacts. *Int. J. Hum. Comput. Interact.*, 39(15):3141–3158, 2023.

[27] J. P. Bowen. The Z notation: Whence the cause and whither the course? In Zhiming Liu and Zili Zhang, editors, *Engineering Trustworthy Software Systems - First International School, SETSS 2014, Chongqing, China, September 8-13, 2014. Tutorial Lectures*, volume 9506 of *Lecture Notes in Computer Science*, pages 103–151. Springer, 2014.

[28] J. P. Bowen and M. Hinchey. Formal methods. In Jr. Allen B. Tucker, editor, *Computer Science Handbook*, chapter 106, pages 106–1–106–25. Chapman & Hall / CRC, ACM, 2nd edition, 2004.

[29] J. P. Bowen and M. G. Hinchey. *Applications of formal methods*. Prentice Hall PTR, USA, 1st edition, 1995.

[30] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Softw.*, 12(4):34–41, 1995.

[31] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[32] J. P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Softw. Eng. J.*, 8(4):189–209, 1993.

[33] A. B. Brown. Oops! Coping with human error in IT systems. *ACM Queue*, 2(8):34–41, 2004.

[34] David A. Carrington, David J. Duke, Roger Duke, Paul King, Gordon A. Rose, and Graeme Smith. Object-z: An object-oriented extension to Z. In Son T. Vuong, editor, *Formal Description Techniques, II, Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'89, Vancouver, BC, Canada, 5-8 December, 1989*, pages 281–296. North-Holland, 1989.

[35] P. P.-S. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[36] E. M. Clarke, O. G., and Doron Peled. *Model checking*. MIT Press, 1999.

[37] K. W. Collier. *Agile analytics: A value-driven approach to business intelligence and data warehousing*. Agile Software Development Series. Pearson Education, 2011.

[38] Overture Community. *The VDM-10 Language Manual*, 2010. `https://www.overturetool.org/documentation/manuals.html`, Last accessed 10 December 2024.

[39] M. Talevi P. Hammant-S. Cresswell C. Gaviao D. North, E. Keogh. What is JBehave?, 2001. `https://jbehave.org/`, Last accessed 10 December 2024.

[40] J. B. Dennis. *Petri nets*, pages 1525–1530. Springer US, Boston, MA, 2011.

[41] J. Derrick and E. A. Boiten. *Refinement in Z and object-Z - Foundations and advanced applications (2. ed.)*. Springer, 2014.

[42] David J. Duke, Bob Fields, and Michael D. Harrison. A case study in the specification and analysis of design alternatives for a user interface. *Formal Aspects Comput.*, 11(2):107–131, 1999.

[43] L. Davide Spano D. Raggett F. Paternò, C. Santoro. MBUI - task models, April 2014. `https://www.w3.org/TR/task-models/`, Last accessed 10 December 2024.

[44] M. S. Farooq, U. Omer, A. Ramzan, M. A. Rasheed, and Z. Atal. Behavior driven development: A systematic literature review. *IEEE Access*, 11:88008–88024, 2023.

[45] C. Fayollas, C. Martinie, P. A. Palanque, Y. Deleris, J.-C. Fabre, and D. Navarre. An approach for assessing the impact of dependability on usability: Application to interactive cockpits. In *2014 Tenth European Dependable Computing Conference, Newcastle, United Kingdom, May 13-16, 2014*, pages 198–209. IEEE Computer Society, 2014.

[46] C. Fayollas, C. Martinie, P. A. Palanque, P. Masci, M. D. Harrison, J. C. Campos, and S. R. e Silva. Evaluation of formal ides for human-machine interface design and analysis: The case of CIRCUS and pvsio-web. In C. Dubois, P. Masci, and D. Méry, editors, *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016*, volume 240 of *EPTCS*, pages 1–19, 2016.

[47] G. Fitzgerald and N. L. Russo. The turnaround of the london

ambulance service computer-aided despatch system (LASCAD). *Eur. J. Inf. Syst.*, 14(3):244–257, 2005.

[48] F. L. Gandon. Ontologies in computer science. In F. Gargouri and W. Jaziri, editors, *Ontology Theory, Management and Design - Advanced Tools and Models*, pages 1–26. IGI Global, 2010.

[49] M. Ghazel, J. Yang, and E.-M. El-Koursi. A pattern-based method for refining and formalizing informal specifications in critical control systems. *J. Innov. Digit. Ecosyst.*, 2(1-2):32–44, 2015.

[50] M. Haesen, K. Coninx, J. Van den Bergh, and K. Luyten. MuiCSer: A process framework for multi-disciplinary user-centred software engineering processes. In P. Forbrig and F. Paternò, editors, *Engineering Interactive Systems, Second Conference on Human-Centered Software Engineering, HCSE 2008, and 7th International Workshop on Task Models and Diagrams, TAMODIA 2008, Pisa, Italy, September 25-26, 2008. Proceedings*, volume 5247 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2008.

[51] M. Haesen, J. Van den Bergh, J. Meskens, K. Luyten, S. Degrandsart, S. Demeyer, and K. Coninx. Using storyboards to integrate models and informal design knowledge. In H. Hussmann, G. Meixner, and D. Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 87–106. 2011.

[52] M. Haesen, J. Meskens, K. Luyten, and K. Coninx. Draw me a storyboard: Incorporating principles and techniques of comics to ease communication and artefact creation in user-centred design. In T. McEwan and L. McKinnon, editors, *Proceedings of the 2010 British Computer Society Conference on Human-Computer Interaction,*

BCS-HCI 2010, Dundee, United Kingdom, 6-10 September 2010, pages 133–142. ACM, 2010.

[53] A. Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990.

[54] H. Rex Hartson, Antonio C. Siochi, and Deborah Hix. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Trans. Inf. Syst.*, 8(3):181–203, 1990.

[55] M. C. Henson, S. Reeves, and J. P. Bowen. Z logic and its consequences. *Comput. Artif. Intell.*, 22(3-4):381–415, 2003.

[56] Andrew Hussey. Formal object-oriented user-interface design. In *12th Australian Software Engineering Conference (ASWEC 2000), 28-30 April 2000, Canberra, Australia*, pages 129–138. IEEE Computer Society, 2000.

[57] RJ Auburn M. Bodell D. C. Burnett J. Carter S. McGlashan T. Lager M. Helbing R. Hosn T.V. Raman K. Reifenrath N. Rosenthal J. Roxendal J. Barnett, R. Akolkar. State chart XML (SCXML): State machine notation for control abstraction, September 2015. `https://www.w3.org/TR/scxml/`, Last accessed 10 December 2024.

[58] D. Jackson. *Software abstractions: Logic, language, and analysis*. MIT Press, revised edition, 2012.

[59] D. Jackson, A. Milicevic, E. Torlak, E. Kang, and J. Near. Alloy. `http://alloytools.org/`, Last accessed 10 December 2024.

[60] S. Jaidka. Formal modelling and analysis of safety-critical interactive systems using coloured petri nets. 2020. The University of Waikato, PhD Thesis.

[61] X. Jia. ZTC: A type checker for z notation user's guide. 1995.

[62] C. B. Jones. *Systematic software development using VDM*. Prentice Hall, 1990.

[63] E-Y. Kang and T. R. Silva. Towards formal verification of behaviour-driven development scenarios using timed automata. In *30th Asia-Pacific Software Engineering Conference, APSEC 2023, Seoul, Republic of Korea, December 4-7, 2023*, pages 612–616. IEEE, 2023.

[64] E. S Kirkendall, K. Timmons, H. Huth, K. E. Walsh, and K. R. Melton. Human-based errors involving smart infusion pumps: A catalog of error types and prevention strategies. *Drug Saf*, 43:1073–1087, 08/2020 2020.

[65] R. Kling. The organizational context of user-centered software designs. *MIS Quarterly*, 1(4):41–52, 1977.

[66] J. C. Knight. Safety critical systems: Challenges and directions. In W. Tracz, M. Young, and J. Magee, editors, *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 547–550. ACM, 2002.

[67] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[68] T. C. Lethbridge and R. Laganiere. *Object-oriented software engineering - Practical software development using UML and java*. MacGraw-Hill, 2001.

[69] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. USIXML: A language supporting multi-path development of user interfaces. In R. Bastide, P. A. Palanque, and J. Roth, editors, *Engineering Human Computer Interaction and*

Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Hamburg, Germany, July 11-13, 2004, Revised Selected Papers, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220. Springer, 2004.

[70] N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214–233, 1995.

[71] C. Martinie, P. A. Palanque, and M. Winckler. Structuring and composition mechanisms to address scalability issues in task models. In P. F. Campos, T. C. N. Graham, J. A. Jorge, N. J. Nunes, P. A. Palanque, and M. Winckler, editors, *Human-Computer Interaction - INTERACT 2011 - 13th IFIP TC 13 International Conference, Lisbon, Portugal, September 5-9, 2011, Proceedings, Part III*, volume 6948 of *Lecture Notes in Computer Science*, pages 589–609. Springer, 2011.

[72] P. Mello, P. Ximenes, R. Lemos, A. L. Bessa, M. I. Cortés, and C. L. Rocha. On the applicability of BDD in a business intelligence project: Experience report. In A. Malucelli and S. S. Reinehr, editors, *Proceedings of the 17th Brazilian Symposium on Software Quality, SBQS 2018, Curitiba, Brazil, October 17-19, 2018*, pages 296–304. ACM, 2018.

[73] A. H. Mughal. Advancing BDD software testing: Dynamic scenario re-usability and step auto-complete for cucumber framework. *CoRR*, abs/2402.15928, 2024.

[74] C. A. Muñoz and R. Butler. Rapid prototyping in PVS. Technical report.

[75] D. A. Norman and S. W. Draper. *User centered system design: New perspectives on human-computer interaction*. L. Erlbaum Associates Inc., USA, 1986.

[76] D. North. Introducing BDD, 2006. https://dannorth.net/introducing-bdd/, Last accessed 10 December 2024.

[77] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[78] P. A. Palanque, J. Vanderdonckt, and M. Winckler, editors. *Human Error, Safety and Systems Development, 7th IFIP WG 13.5 Working Conference, HESSD 2009, Brussels, Belgium, September 23-25, 2009, Revised Selected Papers*, volume 5962 of *Lecture Notes in Computer Science*. Springer, 2010.

[79] J. L. Peterson. *Petri net theory and the modeling of systems*. Prentice Hall PTR, USA, 1981.

[80] N. Plat and P. G. Larsen. An overview of the ISO/VDM-SL standard. *ACM SIGPLAN Notices*, 27(8):76–82, 1992.

[81] J. Rubin and D. Chisnell. *Handbook of usability testing: How to plan, design and conduct effective tests*. John Wiley & Sons, 2008.

[82] J.M. Rushby D.W.J. Stringer-Calvert S. Owre, N. Shankar. Pvs language reference. Technical report, Computer Science Laboratory, SRI International, 2008. https://pvs.csl.sri.com/doc/pvs-language-reference.pdf, Last accessed 10 December 2024.

[83] E. Sanders. *From user-centered to participatory design approaches*, pages 1–7. 04 2002.

[84] E. Sanders and P. J. Stappers. Co-creation and the new landscapes of design. *CoDesign*, 4(1):5–18, 2008.

[85] T. R. Silva. *A behavior-driven approach for specifying and testing user requirements in interactive systems.* PhD thesis, Paul Sabatier University, Toulouse, France, 2018.

[86] T. R. Silva. Towards a domain-specific language to specify interaction scenarios for web-based graphical user interfaces. In M. Winckler and A. Quigley, editors, *EICS '22: ACM SIGCHI Symposium on Engineering Interactive Computing Systems, Sophia Antipolis, France, June 21 - 24, 2022, Companion Volume*, pages 48–53. ACM, 2022.

[87] T. R. Silva. Towards a domain-specific language for behaviour-driven development. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2023, Washington, DC, USA, October 3-6, 2023*, pages 283–286. IEEE, 2023.

[88] T. R. Silva, M. Winckler, and H. Trætteberg. Extending behavior-driven development for assessing user interface design artifacts (S). In A. Perkusich, editor, *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*, pages 485–623. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019.

[89] J. M. Spivey. *Z notation - A reference manual (2. ed.).* Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[90] J.M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 2nd edition, 1992.

[91] D. Travis. Creating usability test tasks that really motivate users. `https://userfocus.co.uk/articles/testtasks.html`, Last

accessed 10 December 2024.

[92] D. Travis. Testscenario. https://www.userfocus.co.uk/articles/writing-usability-task-scenarios.html, Last accessed 10 December 2024.

[93] S. Wolff. Scrum goes formal: Agile methods for safety-critical systems. In S. Gnesi, S. Gruner, N. Plat, and B. Rumpe, editors, *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*, pages 23–29. IEEE, 2012.

[94] J. C. P. Woodcock and J. Davies. *Using Z - Specification, refinement, and proof.* Prentice Hall international series in computer science. Prentice Hall, 1996.