Research Commons

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Characterising Sound Visualisations of Specifications using Refinement

A thesis

submitted in fulfilment

of the requirements for the degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Colin Pilbrow

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2019

Abstract

Visualisations can be used to help analyse, explore, and validate Z specifications. However, if visualisations contain errors or are used incorrectly then they can be misleading and harmful.

The aim of this work is to characterise the soundness of visualisations of Z specifications. We achieve this by using refinement. We look for a refinement relation between a Z specification and its (claimed sound) visualisation. If the relation does not hold, then the visualisation is not sound.

The main type of visualisation we investigate is state diagrams. These diagrams are useful for visualising the state and operations of Z specifications. We look at a variety of state diagrams styles before widening the scope to include μ-Chart visualisations and animated visualisations.

We find that existing refinement methods should not be used for all types of visualisations. To characterise the soundness of partial visualisations we extend the standard rules to include unexamined states and restricted specifications.

Finally, we include examples of the refinement rules being used on our visualisations. This lets us formally show whether each individual visualisation is sound or not.

Acknowledgements

I would like to thank my supervisors, Steve Reeves and Judy Bowen, for the work they have done and the guidance they have provided. I am grateful to the Computer Science department and the University of Waikato. The financial support provided has allowed me to present my research at conferences and has made this work possible. Their encouragement and support has given me the motivation to finish and made the task enjoyable. Special thanks to Andrea Haines and the Student Learning team for hosting the Doctoral Writing Conversations and the invaluable writing retreats. Last but not least, thanks to my friends and family for their continuous support over the years. I could not have done it without them.

Contents

1 Introduction **1**

2 Literature Review **5**

2.1 Why and in what situations are formal specifications important? 5

2.2 Approaches to the validation of formal specifications 6

2.3 Approaches to the visualisation/animation of formal specifications 7

2.4 Visualisation tools for specifications 10

2.5 Errors in Visualisations 12

2.6 Other formal methods 13

2.7 Conclusion 14

3 Z Specification Language **15**

3.1 Z 15

3.2 Bindings 20

3.3 Birthday Book 21

3.4 Using Operations 22

3.5 Further Schema Calculus 22

3.6 Precondition Definition 24

3.7 Introduction to Chaos 25

3.8 Z_C 26

3.9 Summary 29

3.10 Filling Jars 30

3.11 Stopwatch Specification 31

4 Refinement **33**

4.1 Principle of Substitutivity 34

4.2 Preconditions and Postconditions 35

4.3 Operation Refinement (Derrick and Boiten) 36

4.4 Woodcock Operation Refinement 38

4.5 Data Refinement 40

4.6 Summary . . . 45

5 Visualisations **46**

5.1 Visualisation Users . . . 46

5.2 Visualising Z Specifications . . . 47

5.3 Visualisation Soundness . . . 47

5.4 ProZ . . . 49

5.5 Graphical Animation . . . 52

5.6 Conclusion . . . 55

6 State Diagrams **57**

6.1 State diagrams with no current state . . . 59

6.2 State Diagram Visualisation Definitions . . . 68

6.3 Summary . . . 74

7 Restrictions **75**

7.1 Proving Soundness with Standard Methods . . . 76

7.2 Proving Soundness with Restrictions . . . 76

7.3 A New Definition of Soundness . . . 77

7.4 Restrictions . . . 77

7.5 Refinement Comparison . . . 78

7.6 Restriction Strengths . . . 80

7.7 Examined and Unexamined States . . . 83

7.8 Using Special State ⊎ . . . 85

7.9 Lifting and Totalising . . . 87

7.10 $(\overset{\bullet}{U}_R)$ . . . 88

7.11 Refinement . . . 90

7.12 Summary . . . 91

8 Microcharts **92**

8.1 Microcharts . . . 92

8.2 Microchart Examples . . . 93

8.3 The Z Semantics of MicroCharts . . . 95

8.4 Soundness of Visualisations . . . 101

8.5 Changes to Microcharts . . . 102

8.6 Microchart Semantics . . . 104

8.7 Operation Operator . . . 105

8.8 Refinement . . . 109

8.9 Summary . . . 111

9 Animated Visualisations **112**
9.1 Aesthetic Animated Visualisations . . . 112
9.2 Simulation . . . 113
9.3 Animated Visualisation Definition . . . 113
9.4 Refinement . . . 115
9.5 Summary . . . 119

10 Conclusion **121**
10.1 Limitations to our Approach . . . 123
10.2 Future Work . . . 125

Bibliography **127**

Appendices **136**

A Proof Rules **138**

B Proofs for Figure 6.1 **140**

C Proofs for Figure 6.3 **144**

D Proofs for Figure 6.4 **147**

E Proofs for Figure 6.10 **149**

F Proofs for Figure 6.11 **151**

G Proofs for Figure 6.13 **156**

H Proofs for Figure 6.15 **163**
H.1 Initialisation Property . . . 167

I Proofs for Figure 7.3 **169**

J Proofs for Figure 7.6 **172**

K Proofs for Figure 7.7 **177**

L Proofs for Figure 8.2 **181**
L.1 State, Init and Axiomatic Definitions for Simple Sequential Charts . . . 182
L.2 Operation Schemas for Tick . . . 184
L.3 Operation schemas for Pause/Play . . . 189
L.4 Composed Chart TPP . . . 191
L.5 Hiding Operator . . . 192
L.6 Step semantics . . . 194

M Proofs for Figure 8.3 **198**

N Construction of a Sound Animation **203**

List of Figures

3.1 Example tree . . . 27

4.1 An operation shown as a partial relation . . . 38
4.2 Lifting the relation . . . 39
4.3 Totalising the relation . . . 39
4.4 Refinement of the relation . . . 40
4.5 Refinement using downward simulation . . . 41

5.1 ProZ . . . 50
5.2 State Properties . . . 50
5.3 Enabled Operations . . . 50
5.4 History . . . 50
5.5 Increase Specification . . . 51
5.6 2x5 Grid of Images . . . 53
5.7 Graph of State in Jars . . . 54
5.8 Graph of State in Birthday Book . . . 54
5.9 Graph of Explored States in Birthday Book . . . 55

6.1 Empty or not empty . . . 58
6.2 Visualisation of NOperation . . . 60
6.3 State Diagram of Jars . . . 60
6.4 State diagram of birthday book . . . 62
6.5 Birthday Book States where set size is two . . . 63
6.6 Birthday Book States where set size is three . . . 64
6.7 Visualisation of birthday book state . . . 64
6.8 Visualisation of stopwatch . . . 65
6.9 Add, Edit, and Remove A . . . 65
6.10 Precondition labels on states . . . 66
6.11 Alan Turing Visualisation . . . 67
6.12 Small Visualisation . . . 67
6.13 Unexplored Alan Turing Visualisation . . . 68

6.14 State Diagram of IncJump . . . 69

6.15 These diagram states each represent several specification states . . . 70

7.2 Setting a to 1 . . . 78

7.3 Filling and emptying the jar 1 . . . 81

7.4 Filling and emptying the jar 2 . . . 82

7.6 Add, Edit, Remove without Unexplored . . . 84

7.7 Filling and emptying the jar 3 . . . 85

7.8 $(\dot{U}_R)$ 1 . . . 88

7.9 $(\dot{U}_R)$ 2 . . . 89

7.10 $(\dot{U}_R)$ 3 . . . 89

7.11 $(\dot{U}_R)$ 4 . . . 90

7.12 Partial Visualisation of U . . . 91

8.1 Simple Sequential μ-chart . . . 93

8.2 Composed Microchart . . . 94

8.3 Microchart with Local Variable . . . 95

8.4 aMuZed Microchart . . . 95

9.1 ProZ Jars Simulation . . . 114

9.2 ProZ Jars Graphical Animation . . . 117

Chapter 1

Introduction

Formal specifications are objects based on mathematics and logic that are used to specify the requirements and behaviour of software before it is built into code. The goal is to develop a deeper understanding of the system, discover errors, and dispel confusion early in development. The mathematical foundation of formal specifications allow formal analysis and requirements that are reliable and unambiguous.

Various abstraction techniques can be used when creating a formal specification. For example, omission and suppression [94] allow us to hide detail considered to be irrelevant and describe complex behaviour at an appropriate level of detail. Then, we *refine* the abstract design towards a concrete implementation. This changes the focus from describing what the purpose of a system is to how the system works. There are rules that determine if a refinement step is valid. These rules allow us to verify that the implementation meets the requirements of the formal specification.

A specification is invalid if it contains features that are just 'wrong' or do not match the informal requirements. Because the validity depends on the informal requirements, environment and clients, a specification cannot be 'proved' valid. Instead, validation techniques are used that increase the confidence of the specification's validity, such as improving communication with the client and clarifying the logic and mathematics used by the specification.

One validation technique is to visualise the specification. By animating, simulating, or using visual diagrams, the user can improve their confidence in the specification. The visualisations can be shown to clients who do not understand the formal language, but will quickly identify problems in the specification based on the visualisation. Even for the programmers and developers, visualisations can be used to quickly build an informal understanding of a specification while using the underlying mathematical foundation for more formal analysis.

The visualisations can also be incorrect. *Unsound* visualisations can be misleading, improve the confidence in invalid specifications, and even suggest bugs and errors that do

not in fact exist in the specification being visualised. Clearly, before we use the visualisation, we want to verify that it is sound. This leads to our research question: Can we use refinement to characterise the soundness of specification visualisations?

We do not consider visualisation properties such as the aesthetics and usability, instead choosing to focus on soundness. Although it is clear that an unusable or aesthetically displeasing visualisation does not help validate the specification, it is not appropriate to apply formal methods to such properties.

In order to use refinement, we investigate visualisations with formal semantics. However, we also investigate informal visualisations, and show how the visualisation properties that are relevant to soundness can be formalised. Using these techniques we can use refinement to characterise the soundness of a much wider scope of visualisations.

Classic *data* visualisation types such as bar charts and pie charts are not included in this investigation as they are not appropriate for validating specifications. Instead, we will mainly be focusing on state diagrams, although in later chapters we expand beyond this.

Contributions

We show how the soundness of Z visualisations can be characterised using refinement. To achieve this we develop methods for creating formalisations of visualisations in Z. We investigate μ-Chart visualisations, many types of state diagram visualisations, animated visualisations, and believe that these methods are applicable to an even wider variety of visualisation types. We introduce restrictions and show how they can improve the characterisation of soundness for partial visualisations.

Chapter 2

We begin with a literature review. The most relevant work is primarily formal methods and visualisations. We also discuss verification and validation techniques and how they are used to find errors in software, specifications, and visualisations. We also cover a variety of formal specification languages and visualisation tools.

Chapter 3

We continue by introducing the specification language Z. Z is based on set theory and predicate logic, and this formal language is used to describe what a system does unambiguously and without inconsistencies. When in the early stages of software development this is used to carefully analyse the requirements of the system and spot bugs before they are built into the final implementation. In chapter 3 we present the structure and logic of Z specifications, and present example specifications that we will be visualising in the following chapters.

Chapter 4

Once we have an understanding of Z, we introduce the idea of refinement. When developing a system, we begin with an abstract specification and build towards a concrete implementation. Each new step is mathematically related to the previous design, and a set of rules are used to verify that the refinement is valid. We introduce operation refinement and data refinement, which will be used later to characterise the soundness of the visualisations.

Chapter 5

Visualisations of specifications can be used for validation of the specification and communication with the client. There are many different types of visualisation, they can be formal or informal, static or animated, used to visualise the whole specification or used to visualise particular aspects. How the visualisation is interpreted changes depending on the context, content, and previous knowledge. Although visualisations are designed to communicate information more easily than the specification, it still takes time to fully understand the visualisation, and an otherwise sound visualisation can still be misleading if is not fully understood.

In this chapter we give a definition for soundness and decide which visualisations are within the scope of this investigation. We also introduce two methods we will use to formalise visualisations.

Chapter 6

The main type of visualisation we investigate is the state diagram. This covers a broad category of visualisations, suitable for visualising systems with states and operations, such as typical Z specifications. We give examples of a range of state diagrams, and provide examples of the formal semantics that can be used to describe them. The state diagrams we show are examples of visualisations that could be used to help validate a specification. For example, some visualise the entire state space of the specification, while others focus on particular states or use-cases. We end the chapter by presenting the semantics we will be using in the majority of proofs.

Chapter 7

Not all visualisations visualise the entire state space of the specification. In fact, because of problems such as state space explosion, it is often better to create smaller visualisations that focus on a particular problem or other aspect of the specification. We call these visualisations restricted visualisations, because they only visualise a restricted portion of the specification. Refinement relations cannot typically be found between a restricted visualisation and its specification, which would mean that our method labels an entire category

of useful visualisations as unsound. In chapter 7 we show how our method is improved to include these visualisations. We also formally introduce a special state in the state diagrams, the unexplored state $\uplus$.

Chapter 8

To help expand the scope of our study, the final two chapters look at visualisations that are more complex than state diagrams. The first is μ-Chart visualisations. μ-Charts are an evolution of StateCharts, and are similar to UML state machines. We have chosen to investigate these visualisations over other types because the language has a formal semantics given in Z. Visualisations often do not have formal semantics, which can be useful for informal communication. By using a formal visualisation we can further narrow the gap between the formal specification and the informal requirements of the client.

In this chapter we begin by introducing the μ-Chart language, and give some examples of visualisations that use the elements that were not present in the state diagrams we investigated, such as local variables and composition. We then discuss the challenges of applying our methods to a different type of visualisation, and how we overcame them. Most importantly, we introduce the Operation operator to the existing μ-Chart semantics, which allows us to separate the visualisation into different operations.

Chapter 9

In chapter 9 we discuss animated visualisations. Again, we provide multiple examples to demonstrate how these visualisations can be used, and how they differ from static visualisations. It is important to note that the examples shown here are not actually animated. Instead these visualisations are presented as a sequence of images with a brief description.

Appendices

Throughout the thesis we provide many examples of visualisations. The appendices contain the technical proofs for some of these examples. This includes creating formalisations of visualisations and using operation or data refinement to show why a visualisation is or is not sound. In Appendix A we provide some common rules that are useful for our proofs.

Chapter 2

Literature Review

In this chapter we introduce the relevant literature related to this work. We begin with formal specifications and formal methods. Then we investigate how formal specifications can be validated, with a particular focus on visualisations. We then discuss the soundness of these visualisations. Finally, although we are focusing on Z specifications, we look at some of the alternative languages that we could have chosen to use.

2.1 Why and in what situations are formal specifications important?

Formal specifications are a valuable tool when creating safety-critical or complex systems [115, 25]. Formal specifications are used to mathematically describe what a system should do. This has a number of benefits. For example, formalising the high-level requirements of a system at the start of development reduces ambiguity while helping to improve consistency and accuracy of the specification and system [39].

During development, the behaviour of the system can be analysed and proof or model checking techniques can be used to help verification and validation. The formal specification can also be further refined by adding more low-level requirements.

The formal specification can also help when implementing the software system. To help ensure the correctness of the final implementation with respect to the specification, model based testing can be used to automatically generate tests from the specification [110, 85]. Alternatively, the implementation itself can be derived using the specification [40, 81].

Formal specifications have been used to help develop large and safety-critical systems, including train control systems [35] and avionics software [27]. Woodcock *et al.* detail additional examples in their survey of formal methods in industry [115]. Their results showed a generally positive effect on time, cost, and quality when using formal methods.

However, formal methods are not widely used in industry for a variety of reasons [41,

100, 11]. For example, learning how to create and use formal specifications is difficult and the number of sophisticated tools that support formal specifications is limited. For programmers creating small apps or websites the benefits that formal methods provide can seem unnecessary. However, because of the benefits it provides we believe that such methods should be used where possible. For example, a lightweight approach could be used by only formally specifying the critical parts of the system [34, 5].

Formal specifications are an important part of formal methods. These techniques can be very helpful when used during software development.

2.2 Approaches to the validation of formal specifications

Formal specifications are used to find and remove errors early on in development and ensure that the requirements of the system are well known and documented. However, they do not reach the ideal of allowing us to create perfect bug-free software. While we can prove that a program is correct it is only correct with respect to the specification and the specification can have its own errors and mistakes.

Firstly, errors in the specification can cause it to behave differently then intended. This includes misspellings, missing constraints, misunderstandings of the mathematics, and many more mistakes. For software, verification means checking if the implementation has been built right with respect to the specification. For formal specifications, verfication also means checking that the specification has been built right and this can involve checking against earlier specifications. This includes ensuring that the specification is syntactically and grammatically correct. Another problem is ensuring that the specification matches the requirements of the client. Have we understood the requirements correctly or has there been a misunderstanding of what the client wants? Validation of a specification means checking that the specification correctly matches the user's needs. There is a well-known gap between the informal requirements and the formal specification [39] that exists because we cannot formally compare the two. Additionally, it can be difficult to accurately represent many real world aspects, such as the environment where the system will be used.

Verification and validation techniques are used to find these errors [4, 112]. Testing is used to identify defects and to ensure that the specification does what it is expected to. Many of the techniques used in software engineering [82, 6] can also be applied to specifications instead of code. Juha Itkonen, Mika V. Mntyl and Casper Lassenius published a study on manual testing techniques used in industry [53]. For example, programmers will manually explore weak areas, simulate real usage scenarios, and simulate abnormal and extreme situations in order to find problems in the software.

Because specifications are written using precise mathematical language we can also use model checking techniques. These techniques allow us to prove various properties of the specification algorithmically. For example, we can show that a specification correctly prevents the system from entering a state it cannot recover from. These bad states could be the software crashing or, in avionics, planes crashing. The main limitation to this technique is the state-space explosion problem, where the number of states in a complex system can become too large to analyse [1].

We can also use *refinement* [29] to ensure that our specification is free of errors. This will be discussed in detail in chapter 4. It essentially allows us to make changes to our specification, for example by adding more low-level requirements, and then prove that the modified specification still matches the original.

To help ensure that the specification is correct with respect to the client's requirements, communication is important. Because the specification is written formally it can be difficult for the client to understand. To help avoid this problem, there are various approaches that may be taken. For example, the clients can use a checklist or informal requirements document when asking if important requirements have been formalised correctly. Alternatively, a walk-through can be performed where what is described in the formal specification is explained to the client [105], for example, by simulating a real usage scenario.

By using these techniques and others we can verify and validate specifications. This will help find errors and ensure that we are creating a specification that the client will be satisfied with. We can then use the specification with increased confidence when developing safety-critical systems.

2.3 Approaches to the visualisation/animation of formal specifications

We are investigating visualisations of specifications. Visualisations help aid and assist the use of formal methods. Visualisations can help assist verification by showing where and why a specification has not been built correctly and help assist validation by making it easier for clients to see when the specification does not meet their needs.

Card, Mackinlay, and Shneiderman [22] have summarised [84, 12, 50] how visualisations can amplify cognition in six major ways:

1. By increasing the memory and processing resources available to the users;
2. By reducing the search for information;
3. By using visual representations to enhance the detection of patterns [99];
4. By enabling perceptual inference to make some problems obvious [63];

5. By using perceptual attention mechanisms such as appearance or motion;
6. By encoding information in a manipulable medium [23].

The visualisations we use can incorporate images, icons, symbols, and sketches as part of the visual form. According to Robert L. Harris [43], some reasons for using these elements are:

- To make the document more interesting and appealing;
- To make the material more understandable to a greater number of people. (The use of pictures sometimes helps overcome differences in language, culture, and education.);
- To improve communication in situations where the appearance of an item is better known than its name or number;
- To facilitate easier reading of a chart or graph by including information to orient the reader that otherwise might have been shown in a legend or note.

There are many types of visualisation, each with their own strengths and weaknesses. We are focusing on diagrams which are used for visualising *nonquantitative interrelationships*. Flow charts are an example of such a diagram. Some reasons to use flow charts (and other types of diagrams) are listed below [43]:

- Describe processes, ideas, networks, etc., particularly complex and abstract ones.
- Define, analyze and better explicate processes, procedures, sequences, etc.
- Improve communications.
- Help to clarify ideas.
- Aid in trouble shooting.
- Serve as a tool in planning and forecasting.
- Reduce misunderstandings and conserve time.

The category of diagrams encompasses a wide variety of visualisations with different purposes and aesthetics. For example the Unified Modeling Language (UML) includes Activity, Sequence, Use Case, and Statechart diagrams [31].

Visualizing Data [26] shows the power of using visualisations for analysing experimental data. It provides many examples of analysts that did not use visualisations reaching incorrect conclusions.

Next we look at some of the ways visualisations have been used to help improve formal method techniques [33].

Dulac, Viguier, Leveson, and Storey have investigated the use of visualization in formal requirements specification [32]. They note "while automated analysis tools can find some types of errors, detecting many of the most serious semantic errors requires human expertise." To help aid this human expertise they propose nine principles for designing visualisations of formal specifications. For example, visualisations should support the most difficult mental tasks and match the task being performed. For example, a graphical overview of the entire specification can help support top-down review. However, this visualisation may hide some dependencies and details. Different notations should be used to support alternative problem-solving strategies, highlight hidden dependencies, and allow investigation into different parts of the system.

In [76] Mathijssen and Pretorius create a formal specification of an automated parking garage and use simulation and visualisation to validate the system. This allowed them to identify and correct a number of problems early on, including issues that may not have been noted otherwise.

An initial user study comparing the readability of a graphical coordination model with Event-B notation [59] tested the effectiveness, efficiency and satisfaction of a *Peer Model* visualisation. The participants showed a preference for the visualisation, which allowed them to understand the system faster.

Testgraphs are directed graphs that partially visualise the specification [78]. States and transitions that are of interest are included in the testgraph. The testgraphs can be used to derive test sequences and animations [80].

2.3.1 Animated and static visualisations

A diagram printed onto paper or drawn on a whiteboard is a static visualisation. Computers allow us to create visualisations that contain more information than can be seen at first glance.

Including interaction and animation in a visualisation can help the user to interpret the specification and to amplify cognition [22]. An animated visualisation can provide additional details about the state by clicking or hovering over elements of the visualisation. The current state of a state diagram can be highlighted as the specification is simulated. To help visualise larger specifications users can apply projections to the visualisation [55] that divide the specification into parts and allow the user to visualise each part in turn.

Animating the visualisation helps the user to interpret the specification. An animation can allow a non-formal methods expert to interact with the specification and test that it meets the expected goals [91]. Animating the visualisation helps the user to further understand how particular operations change the system and identify high level structural errors [72].

A major problem of static visualisations is that their usability decreases as the number of states and transitions increase. A Z specification can have infinite states and naively trying to visualise this will create an unusable visualisation. Animated visualisations are a way to reduce the number of states shown on the screen at any one time.

For animated visualisations the user can use a different method of analysing a large specification: exploratory analysis [108]. Animated visualisations can allow the user to focus on a particular sequence of operations of their choice, exploring the full state space of the specification gradually.

Of course, animated visualisations also have their shortcomings. Many of these problems are shared with static visualisations and can be addressed with visualisation design. Firstly, it is difficult to provide examples of animated visualisations in a static environment. Secondly, animated visualisations are less effective than static visualisations at providing the "whole picture" view of the specification [83]. To help the user gain a complete understanding of the specification it is best to pair the animation with accompanying text or additional views of the specification. A consequence of the visualisation not communicating the "whole picture" effectively is that when the user is exploring the specification they may become lost or confused. The user may miss parts of the specification or be unsure as to whether they have sufficiently explored the system to validate any beliefs about the system.

Although animated visualisations try to avoid the state space explosion problem by focusing on particular states they do not entirely succeed. While a large number of states are not shown at once they nevertheless still exist and completely exploring an animated visualisation that has a large state space will take a long time.

Exploration analysis and nondeterminism do not work together smoothly. In Z specifications nondeterminism in an operation implies that regardless of the result the user will be satisfied and unable to meaningfully distinguish between the possible outcomes. As such, nondeterministic transitions in a graphical animation could go unnoticed by the user. One approach to handling nondeterminism is to split nondeterministic operations into multiple operations that the user can choose between.

2.4 Visualisation tools for specifications

There are a number a tools that can be used to visualise formal specifications. These tools can automatically create a visualisation or be used to aid the creation of custom or specific visualisations.

We start with ProB [67, 68], a visualisation tool and model checker. ProB has a number of features. For example it can be used to simulate a specification. This allows us to perform a walk-through of the system. Diagrams visualising the state space of the specification can be automatically generated. These diagrams can be developed further to help assist human

analysis. For example, by using projection diagrams and domain-specific visualisations [62]. Projection diagrams are used to help reduce the size of the state space by both highlighting particular parts of the system and hiding irrelevant parts [18]. Images and other pictorial elements that are related to the system can be included to create domain-specific visualisations. This is particularly helpful for non-formal methods experts to help understand the specification. Although ProB was originally designed for B, it also supports Event-B, CSP-M, TLA+, and Z. ProZ is a extension of the ProB animator and model checker designed to support Z specifications. We discuss ProZ in section 5.4.

B-Motion Studio is another tool used to generate domain specific visualisations for Event-B [60] and CSP models [61]. These visualisations help the user develop a better understanding of the corresponding specification and can be used to identify inconsistencies or unexpected behaviours within the specification. A user study about the B-Motion Studio visualisation of a real-life industrial example had positive results [106]. Participants said they would use these visualisations for "prototyping and validation, impact analysis, replay of execution logs and predefined sequences".

Jaza is an animator for the Z specification language. It has been used to help validate UML and SecureUML class diagrams that have been converted into Z [66, 65, 93]. The Community Z Tools (CZT) project [75, 74] includes the ZLive animator, which is the successor to the Jaza animator [109]. Possum is an animator for the SUM specification language [44, 79]. SUM is closely related to Z and so Possum can also be used to animate some Z specifications.

Akram Idani and Nicolas Stouls provide methods for creating visualisations from B formal models [52]. They have implemented these methods using the *GeneSyst* tool and performed a user study to evaluate the understanding of specifications when diagrams are provided. They found that diagrams significantly reduce the error rate for some questions.

Metric Temporal Logic (MTL) can be used to formalise cyber-physical systems. A graphical formalism for visualising MTL specifications has been proposed [49]. Users can also use this graphical formalism to create formal MTL specifications using the ViSpec tool [48].

Z specifications can also be visualised using a combination of UML diagrams [58]. This is done to improve the readability and understandability of the specification. Here, UML Class Diagrams are used for the state schema, Contract Boxes are used for the operation schemas, and invariants are visualised using Constraint Diagrams. This helps show the importance of matching the specification with the appropriate visualisation types.

Automated Abstractions for Contract Validation [24] introduces the CONTRACTOR tool for automatically generating Finite State Contract Abstractions. They have applied this tool to an industrial case study and the resulting visualisations "led (them) to discover previously unknown inconsistencies or omissions in real-life specifications". They also pro-

vide a set of guidelines and heuristics that aim to help improve validation when using their visualisations.

Z specifications can be translated into executable languages like Haskell [102] and then animated. For example, PiZa is a tool that translates Z specifications into Prolog [46, 104, 101]. This allows users to find errors that would otherwise be difficult to spot and which will not be detected by type checkers.

Visualisations are a valuable tool and when used correctly they can help users understand and debug formal specifications.

2.5 Errors in Visualisations

Validation is important for ensuring our software and specifications are correct. However, errors in our validation techniques can provide misplaced confidence in our system and delay the discovery of problems.

A bug in a unit test can also hide a bug in the software. However, it is already expected that testing will not find every error [117]. A visualisation that is drawn incorrectly or a misleading specification walk-through can cause a miscommunication with the client or a misunderstanding of what the correct system should be.

There are many types of visualisation errors, ranging from graphical to semantic. Visualisations can be difficult to read, overly complex or deceptive [107]. A visualisation that relies on animation or interaction can hide important information.

We have shown that there are many types of visualisation in section 2.3. Many of these share common graphical elements, however the underlying semantics of these visualisations are different. For example, for different visualisation types, a circle can represent a particular system state, an abstract combination of different states, an operation, or input. Problems can arise if the semantics for the visualisation are not known or misunderstood. For example, the case study of the Peer model showed that some users misinterpreted the graphical representation of the invariant assertions and so gave incorrect answers in a questionnaire about the model [59]. Some visualisation types do not have complete formal semantics, such as UML [64]. So, the exact meaning of these visualisations can be ambiguous or unclear.

Another possible problem is that the visualisation does not match the specification. This could be caused by a mistake while drawing or because the visualisation is incomplete. The specification may have been updated and the changes not reflected in the visualisation. An error in the formal specification can mean the visualisation matches the informal requirements but not the formal.

There are some techniques that can be used to help reduce errors in visualisations. For example, visualisations should be clear and intuitive [36]. There should be minimal semantic distance between the visualisation and the users model of how the system works [51], for

example, by using notation that the user is familiar with and not changing existing notation. Computer generated visualisations greatly reduce the human error that can be introduced when creating a visualisation by hand.

How important is it to prove the soundness of visualisations? Like software tests, a large number of visualisations can be created, which helps reduce the impact when one is found to be incorrect. As the specification is developed visualisations will be altered and outdated which can cause a previously sound visualisation to become unsound. Visualisations are not the only method of validation and as such are not solely relied upon to find errors. An obviously unsound visualisation will be quickly discarded, replaced, or fixed, but not used seriously to help validation.

Despite this however, it can still be useful to prove the soundness of visualisations. For example, unlike tests which are created in bulk to cover a wide base of the software, some visualisations can be particularly important. This could be a visualisation that covers the entire state space of the specification, or a visualisation that shows a particularly complex constraint in a simple graphic. Ensuring that this visualisation is sound early is important, before the user can reinforce an unsound mental model of the specification by referring to the visualisation repeatedly as a guide.

Research that focuses on the soundness of visualisations is limited. As Yaman Barlas says: "Validity of the results in a model-based study are crucially dependent on the validity of the model. Yet, there is no single established definition of model validity and validation in the modeling literature." [9] We address this gap in knowledge by formally characterising the soundness of Z visualisations and showing how this can be used identify unsound visualisations.

2.6 Other formal methods

Although we are focusing on the Z specification language there are many alternative languages that could have been chosen such as Alloy [55], OCL, VDM [14, 13], StateCharts[42], or B. We discuss below a range of languages from textual to graphical.

B [3] is a specification language related to Z that is more focused on refinement. Event-B [2] is an extension of this language. Alloy [54] is influenced by Z, however, it is based on first order logic. It also includes graphical elements that can express the state of the system. The Object Constraint Language (OCL) [21] is a textual language that started as a complement for UML. OCL is more expressive and can specify details more precisely than UML alone. SOFL (Structured Object based Formal Language) [73] is a specification language that includes a mixture of natural languages, graphical notation, and formal notation. A condition data flow diagram (CDFD) is a directed graph that describes the dynamic structure and gives a graphical view of the system. Petri nets [98] use a formal graphical notation as their

specification language. Petri nets are used for distributed systems and other systems that involve concurrent execution.

We chose a textual language as this allows us to investigate visualisations that are not already part of the language. Out of these we chose Z as it was most familiar to us. The methods we use can be applied to other formalisms so a different choice of language could be investigated. Additionally, it is possible to translate a specification that has been written in one language to another [38, 81, 16].

2.7 Conclusion

In this chapter we have looked at the literature related to our work. Our first main focus is formal specifications. Here, precise mathematical language is used to specify exactly what a system should do. We referenced several industry case studies where this technique has been used to save time and money. Then we looked at verification and validation and how we can find problems in our formal specifications. Our next main focus is visualisations. These are widely used to support a variety of tasks, including formal methods. Visualisations can help everything from the creation and analysis of specifications to communication with clients. Because of this some formal languages include graphical notation while others use visualisation, animation, and simulation tools.

Visualisations can also have errors. We discussed how and why this is important in section 2.5. Characterising the soundness of a visualisation has not been done outside our own work. We have previously investigated the soundness of state diagrams [88] and μ-Chart visualisations [87]. We discuss this in more detail in chapters 6 and 8. Finally in this chapter, we introduced a variety of formal specification languages we did not use. We will be focusing on Z specifications which are the topic of the next chapter.

Chapter 3

Z Specification Language

Before building a system it is important to first ask what the purpose of the system is. A specification defines what a system should do. An informal specification can be as simple as 'I want a program that can record the birthdays of my friends'. However, informal specifications can be ambiguous and imprecise, so we can also create formal specifications that are written in formal specification languages, such as Z. These languages are based on standard mathematical notation, including set theory and predicate logic. This lets us create unambiguous specifications, and allows us to use formal verification techniques such as model checking, invariants, and theorem proving to help prove various properties of the specification. However, specification languages are more abstract than programming languages. A specification can be nondeterministic and can include sets of infinite size. This means a typical specification cannot be executed. Instead they are used as the basis to create valid programs by using techniques such as refinement and building test suites.

3.1 Z

In this thesis we will be using the Z specification language. Z is a formal specification language used to specify and model computer systems. Z was developed by a team at the Programming Research Group of Oxford University. Many books have been written about Z, from the notation to case studies. We therefore give a very brief introduction and further information can be found in these texts and others [114, 71, 56, 8].

- We begin by introducing schemas. Schemas are used to specify the state space and operations of the specification.
- Z is a strongly typed language. In section 3.1.2 we discuss how to create and use types.
- Schema calculus can be used to define and combine schemas. We introduce some of the operators we use most often, such as schema inclusion and schema disjunction.

- Z specifications tend to follow similar layouts. We discuss this in subsection 3.1.4.
- Then, we look at bindings, which are collections of observations, and the elements of schemas when schemas are viewed as sets.
- In section 3.3 we present an example specification of a birthday book.
- Then, we look at how Z specifications work in a larger environment. We discuss how specification operations are "used" to change the state of the system, and how a sequence of operations creates a program.
- Then, we look at operation preconditions and postconditions and provide the definitions of some more complicated schema calculus that will be used to build the formal definition of preconditions.
- Next, we look at what happens when we use an operation outside its precondition, and the $\perp$ state.
- Finally, we introduce the logic of Z, Z_C, which we will be using later in the thesis.

3.1.1 Z Schemas

A Z schema has two main parts, the declarations and the constraints, as well as an optional name.

The schema can also be defined horizontally to save space.

$Name \mathrel{\widehat=} [\, Declarations \mid Constraints \,]$

The declarations section lists the observations. Each observation has a name, a type, and possibly some decoration. The decoration is a terminating character after the name of the observation that distinguishes (by convention) the sort of observation. The ? decoration indicates that it is an input observation. ! indicates output observations. The $'$ decoration is found in operation schemas and indicates the decorated observation contains the value of the original observation after the operation is used. For example, the following schema describes an operation called *Increase* that takes an input called *amount*? and increases n by that amount.

Increase

$n, n' : \mathbb{N}$

$amount? : \mathbb{N}$

$n' = n + amount?$

In this example, our observations are n, n', and $amount?$, where $amount?$ is an input observation and n and n' are the before and after observations of the number we are increasing. Observations are local to the schema they are introduced in. Global constants can also be introduced that all schema have access to and these are introduced by *axiomatic* definitions.

The constraints section adds constraints to the observations that have been declared. In a state schema, the constraints section is used to restrict the state space. For example, we may specify that the value of an observation cannot go beyond some maximum value. If the afterstate values are restricted, we instead say the afterstate values are set. For example, in the earlier *Increase* specification, n' is set to the sum of the old value of n plus $amount?$. While the constraints section is simply a proposition, to help make the schema more readable we can split the proposition into different lines, where each new line in the constraints indicates a hidden $\wedge$ (logical conjunction operator).

3.1.2 Types

Z uses typed set theory. A type is a group of similar common elements, and when we introduce an observation we also define what type it has. For example, $n : \mathbb{N}$ defines n as a natural number. This helps avoid certain mathematical paradoxes found in logic that is not strictly typed, while also improving automated checks and proofs about the specification. However, its primary use is to help clarity. If n was introduced without giving its type explicitly, we would be missing information that could be valuable to our understanding of the specification. Perhaps we could infer that n was a number, based on how it was used, but we would not immediately realise that it can never be negative.

The core language of Z only includes the built-in type $\mathbb{Z}$, which represents the set of all integers. The mathematical toolkit is an extension to the core language that includes additional types such as $\mathbb{N}$, the natural numbers, as well as a large number of operations. If additional types or operations are required, or we choose not to use the toolkit, we may define our own. Types are built in a way that maximises readability and understanding of the specification.

Basic types (also called given sets) are declared by writing their name in square brackets, typically in all caps. For example, the set of all names can be defined as follows:

$[NAME]$

We can now declare a variable $n : NAME$ which can be read as 'n is a $NAME$'. There are no further restrictions on the form of $n : NAME$. For example, we do not need to specify the length of the name or how it is written. This is a useful example of abstraction as we can use n while leaving this information unspecified. We can add this information later, when it becomes important.

Alternatively, we can introduce a type by listing its elements. Here we give the general format and a simple example:

$$freeType ::= element_1 \mid element_2 \mid \ldots \mid element_n$$
$$DIRECTION ::= up \mid down$$

These are then used in the same way as the basic types. If $d : DIRECTION$, then the value of observation d is either up or $down$.

3.1.3 Schema Calculus

Schemas can be combined and composed using schema calculus. We will briefly cover the schema calculus used in this thesis. However, many of the details are omitted, such as when an operation is illegal and how decorations are handled. We begin by introducing simple schema calculus, while more complex schema calculus is introduced later in the chapter.

We will be using schema *ExampleState* for the following examples:

ExampleState

$n : \mathbb{Z}$

$n < 100$

We can decorate the schema name with a prime, this primes all observations in the schema. This is used to indicate *ExampleState* after some operation has been used. $ExampleState'$ is the schema:

$n' : \mathbb{Z}$

$n' < 100$

The delta notation is used to include both the before and afterstates in one schema. This is very useful when making operation schema that change the state. The following schema is $\Delta ExampleState$:

$$n, n' : \mathbb{Z}$$

$$n < 100$$
$$n' < 100$$

We can include schemas within other schemas, using *inclusion*. Here, the operation schema *Increase* includes the observations of $\Delta State$, and the constraints are conjoined to the predicate part.

Increase

$$\Delta ExampleState$$
$$amount? : \mathbb{N}$$

$$n' = n + amount?$$

If we expand this schema we get the following:

$$n, n' : \mathbb{N}$$
$$amount? : \mathbb{N}$$

$$n < 100$$
$$n' < 100$$
$$n' = n + amount?$$

Schema conjunction and schema disjunction are also used similarly to the logical operators. The schema that is created includes the observations of both schema being combined, while their predicates are conjoined or disjoined, respectively.

3.1.4 Specification Layout

There is an established strategy used to organise information into schemas and schemas into specifications. Although Z specifications can be written in different styles, traditionally Z specifications are constructed as a state space with a set of operations that operate on the state space. The parts of Z specifications are typically written in a particular order. First, any given sets and global constants are provided with a description of how they will be used in the specification. Then, the state schema is presented. This schema includes the observations we will be using throughout the specification. An initial state schema that defines the initial state of the system is presented. This is specified using the observations of the state space with some additional predicates used to set the system to the initial

values we want. Operations of the system can be used to change the state and output values. For each operation, a schema is created. Schema calculus can be used to combine schemas, so smaller operation schemas can be built and then joined together afterwards. For example, any error handling can be defined in its own schema, and combined with the main operation schema later in the specification. Lastly, Z specifications do not consist entirely of mathematical notation. Instead, they include a large amount of natural language that informally describes the purpose, meaning and significance of the formal specification. The informal commentary helps relate the mathematics to real world examples and is vital to building a clear and understandable specification [116].

3.2 Bindings

The schema can also be considered to be a set of bindings. A binding is a structure that associates observation names with values. An example of a small binding:

$$\langle n \mapsto 0 \rangle$$

An example of a binding in the *Increase* schema:

$$\langle n \mapsto 1, amount? \mapsto 3, n' \mapsto 4 \rangle$$

This binding is well-formed because the names and types of the observations match the *Increase* schema. Because the observations also satisfy the constraint of *Increase*, this binding is in the schema.

The following binding is not in *Increase* because it is not well-formed, the names of the observations in the binding do not match the schema.

$$\langle n \mapsto 1, m \mapsto 4 \rangle$$

This binding is not in *Increase* because the values of the observations do not satisfy the *Increase* constraints.

$$\langle n \mapsto 1, amount? \mapsto 0, n' \mapsto 0 \rangle$$

The order of the "observation value" pairs in the binding does not matter, but to help readability, we have first listed the initial observations, followed by the input observations, and finally the afterstate observations.

If, and only if, the values in the well-formed binding cause the proposition in the constraints to be true, the binding will be included in the schema. A schema may contain an infinite number of bindings.

3.3 Birthday Book

We will now look at an example specification. A commonly used example of a Z specification is a specification of a birthday book [103]. We want to remember the birthdays of our friends by storing them in a book.

We have two basic types, names and dates. These sets contain all names and all dates, respectively. How they are represented is not relevant to how the birthday book works, and does not need to be included. This allows us to work at a higher level of abstraction, without worrying about strange characters or leap years.

$$[NAME, DATE]$$

The state of the birthday book is specified using two observations. *known* is a set of the names we have recorded in the birthday book. *birthdaybook* is a partial function that maps names to dates. So each name in the birthday book is associated with the birthday of the person with that name.

State

$$birthdaybook : NAME \pfun DATE$$
$$known : \mathbb{P}\, NAME$$

$$known = \operatorname{dom} birthdaybook$$

We start with an empty book. The number of names we initially know is zero. *Init* does not appear to have any *birthdaybook* constraints, so this schema looks nondeterministic. However, the constraint in *State*, included in *Init*, implies *birthdaybook* is empty too.

Init

$$State$$

$$known = \varnothing$$

If we get a new friend we can add their name to the date in the book that is their birthday. We input a name and a date, and if the name is not already in the birthday book, we add it. This is done formally using set union.

AddFriend

$$\Delta State$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$$

We can remove a name from the birthday book. The name being removed is input, and if it is in the book, then we remove it using domain restriction.

RemoveFriend

$\Delta State$

$name? : NAME$

$name? \in known$

$birthdaybook' = \{name?\} \ntriangleleft birthdaybook$

If we need to, we can also edit entries in the book. This changes the date associated with name being input to the date being input. This can only be done if the name is already in the book, and uses $\oplus$ to override the old entry.

EditFriend

$\Delta State$

$name? : NAME$

$date? : DATE$

$name? \in known$

$birthdaybook' = birthdaybook \oplus \{name? \mapsto date?\}$

3.4 Using Operations

Operations can be viewed as binary relations over a state space relating a beforestate to an afterstate. Alternatively we can say that with a given state and inputs, we can apply the operation and find related afterstates and output. It is possible that multiple afterstates are related to the same start state. This is because Z operations can be *nondeterministic*.

We can create an imaginary *program* that applies the *Init* schema, followed by a sequence of operations with inputs. The result of a program is the outputs produced by the operations used, and the final resulting state. If an operation in the program is nondeterministic, then the same program may result in different outputs. An imaginary *observer* of the specification can execute programs and observe the output and states.

3.5 Further Schema Calculus

Consider the *RemoveFriend* example we provided previously in section 3.3. This operation is used to remove a name from the birthday book. The relation that this operation defines is

only partial, as when the birthday book does not include the input name, we do not specify what the afterstate should be.

The precondition of an operation characterises the states and inputs which relate to some afterstate and outputs. For example, the precondition of the *RemoveFriend* operation is that the input name is included in the book. We discuss what happens if the name is *not* included in section 3.7, and give a formal definition of the precondition of an operation in section 3.6, but first we provide some more schema calculus that will be useful.

Similarly to first-order logic, we can quantify over schemas. This can be used to express universal and existential properties of schemas. Quantification removes a given observation x from the declarations of the schema, and quantifies the predicate over x.

Existential Quantification

We are given a schema S == $[x : X;\ Decl_S \mid pred_S]$, where the set of observations in $Decl_S$ does not contain $x : X$. The existential quantification over x in S is

$$\exists\, x : X \bullet S == [Decl_S \mid \exists\, x : X \bullet pred_S]$$

This can also be called *hiding* the observation x, because x is removed from the declarations. Instead of hiding a single observation $x : X$, we can also hide all observations in a given schema U from another schema S. We are given a schema S == $[U;\ Decl_S \mid pred_S]$, where the set of observations in $Decl_S$ does not include any observations in U, and $Decl_U$ has observations $x_0 : X_0, x_1 : X_1, \ldots, x_i : X_i$. The existential quantification over U in S is

$$\exists\, U \bullet S == [Decl_S \mid \exists\, x_0 : X_0, x_1 : X_1;\ \ldots, x_i : X_i \bullet pred_U \wedge pred_S]$$

For example, observe the following state schema, which has two observations.

```
┌─ State ──────────────────────
│ x : ℕ
│ y : ℕ
├──────────
│ x < 10
└──────────────────────────────
```

The *Operation* schema below includes the observations in *State*.

```
┌─ Operation ──────────────────
│ ΔState
├──────────
│ x' = x + 1
│ y' = x
└──────────────────────────────
```

We will hide the *State* schema observations, leaving only the afterstate observations of *Operation* in the declaration section. The following schema shows $\exists\, State \bullet Operation$:

$$
\begin{array}{|l}
x', y' : \mathbb{N} \\
\hline
\exists\, x, y : \mathbb{N} \bullet x < 10 \\
x' < 10 \\
x' = x + 1 \\
y' = x
\end{array}
$$

Universal Quantification

We are given a schema S == $[x : X;\ Decl_S \mid pred_S]$, where the set of observations in $Decl_S$ does not contain $x : X$. The universal quantification over x in S is

$$\forall\, x : X \bullet S == [Decl_S \mid \forall\, x : X \bullet pred_S]$$

Thus, the resulting schemas $\exists\, x : X \bullet S$ and $\forall\, x : X \bullet S$ contain all observations of S but x. This will be shown to be useful in the following definitions for preconditions and refinement.

We can also quantify schema U over S to remove the observations of U from the declarations of S.

3.6 Precondition Definition

The precondition characterises the states and input which relate to an afterstate when the operation is applied. It can also be called the states where the operation is applicable, or where the operation is able to be used successfully.

The precondition $\mathbf{pre}\, Op$ of operation $Op == [\Delta State;\ In;\ Out;\ \mid pred]$ with state $State$, input observations In and output observations Out is defined as:

$$\mathbf{pre}\, Op = \exists\, State';\ Out \bullet Op$$

This hides the afterstate and outputs of Op, resulting in a schema that only includes beforestate and input observations. For example, $\mathbf{pre}\, AddFriend = \exists\, birthdaybook';\ known'; \bullet AddFriend$. This can be simplified using the one-point rule and other equivalences to the schema $[State;\ name? : NAME;\ date? : DATE \mid name? \notin known]$. This matches with what we expected, the $AddFriend$ operation can be used successfully when the name is not already in the book.

If the precondition of an operation is **true**, then the operation is *total* and can be used to get the 'expected' result in any beforestate with any valid input. That is, the state and input observations can have any value of the correct type. Otherwise, it is a *partial* operation, as it is not applicable for all inputs and states in the domain. The $AddFriend$ operation is partial because it does not specify the afterstate when $name? \in known$.

3.7 Introduction to Chaos

Allowing operations to be partially specified is useful for a number of reasons. We may specify the important parts of the system. Anything that we have not specified is taken as not important for the system to work and could be implemented in any way without affecting the target system. Since we have not specified what we want to happen, we are happy with any result. The initial specifications we create are highly abstract, and through refinement we add more details. By allowing partial specifications, we are able to create more abstract specifications that focus on the important parts of the system.

However, when we allow partial specifications, we must ask the question: what happens when an operation is used in a state where the behaviour has not been specified, i.e. when the precondition is false? The two most common interpretations are blocking and chaos.

The blocking interpretation states that we cannot use an operation where the precondition is false. For example, we could not use *AddFriend* when the input name is already in the book. If it is not possible to use the operation, then it is not possible for the operation to cause any nasty undefined side effects. However, the question we asked was what happens when the operation is used outside its precondition, and 'actually it wasn't used, because it could not be used' can be considered to be an unsatisfactory answer.

The standard interpretation of Z operations is the chaotic or *contractual* interpretation. If we do not want or cannot block the operation from being used, how will it behave? Chaotically. When the operation is used outside its precondition, the afterstate values could be anything. When using the operation outside its precondition, the values might stay the same, they could follow earlier specified behaviour, or result in any other state, nondeterministically. This includes a state called $\perp$ (pronounced bottom), which is a state that represents the system reaching a state that is chaotic, blocked, unsafe or broken. $\perp$ is a state we want to avoid. However, we include it because it is possible that when using an operation outside its precondition we could cause the system to break.

Previously we discussed how we should handle operations being used outside of their preconditions. However, it is possible to create an operation that specifies what happens to some of the observations but not all. Consider a state schema with two observations, x and y, that are natural numbers. We could create an operation called *IncreaseX* that simply specifies that we increase the value of x. However, when this operation is used, how does the value of y change? We have not specified this within the operation. Again, there are two common interpretations. The first is to do nothing. The value of x will change as we have specified it, while the value of y will not change. Alternatively, if we use a more chaotic interpretation, the value of y could change to any natural number when we use this operation. The difference between the two options here is different to the blocking and chaos that we looked at previously. This is because "do nothing" is a possible refinement of

the chaotic approach, while we cannot refine a chaotic operation into a blocked operation. Because of this, the chaotic approach is standard, as it allows us to be more rigorous. We could choose to refine the operation into "do nothing", but we could also refine the operation into something more suitable, depending on the circumstances.

Z operations can be nondeterministic or partially defined. These types of abstraction have their advantages when first creating a specification. However, we want the final specification to have no uncertainty. So, we improve our operations by removing non-determinism and making the operations total. We develop our abstract specification toward a more concrete specification using *refinement*. We discuss this further in chapter 4.

3.8 Z_C

Z_C is an extension of higher order logic for Z. It includes the addition of schema types. More information about Z_C and the logic of Z can be found in [45]. We define the types of Z as follows.

$$T ::= \Upsilon \mid \mathbb{P}\, T \mid T \times T \mid [\ldots z^T \ldots]$$

These types are free types, power sets, pairs, and schema types. Note that in this metalanguage we use superscript to denote type.

Free types are given by equations with the following form:

$$\Upsilon ::= \ldots \mid c_i \langle\!\langle \Upsilon_{ij} \rangle\!\rangle \mid \ldots$$

Υ is the name of the free type. c_i are constructors which inject values from the set Υ_{ij} to Υ. Υ_{ij} are expression types, which may include Υ (permitting recursion). $\langle\!\langle \Upsilon_{ij} \rangle\!\rangle$ may be omitted, if so, c_i is simply a constant of type Υ rather than a constructor.

We provide three examples of free types. The first example is simple, and introduces five constants with type $VOWELS$.

$$VOWELS ::= a \mid e \mid i \mid o \mid u$$

The following examples are recursively defined free types. In the first example, we define $\mathbb{N}$ as a free type.

$$\mathbb{N} ::= zero \mid succ\langle\!\langle \mathbb{N} \rangle\!\rangle$$

Next, we give the type of a tree, where each node has contains a natural number. Additionally, each node is either a leaf or branches in two.

$$TREE ::= leaf\langle\!\langle \mathbb{N} \rangle\!\rangle \mid branch\langle\!\langle \mathbb{N} \times TREE \times TREE \rangle\!\rangle$$

leaf zero is the simplest value with type $TREE$. A more complex example is the following value, which we can also see drawn in Figure 3.1. Here numbers are used to represent

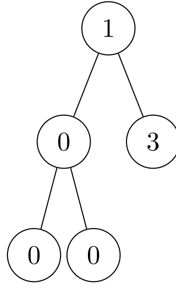


Figure 3.1: Example tree

elements from $\mathbb{N}$.

$$branch\,(succ\ zero, branch\,(zero, leaf\ zero, leaf\ zero), leaf\ succ\ succ\ succ\ zero)$$

Schema types are the types for bindings and have the form $[\ldots z_i^{T_i} \ldots]$. This is an unordered sequence of observations z_i with types T_i.

However, note that the type of a given binding will not be clear without also knowing the schema it is an element of. For example, consider the binding $\langle\, x \mapsto 0, y \mapsto 2\,\rangle$. Although we know the form of this binding is $[\, x^{T_i}, y^{T_j}\,]$, we do not know what the exact types of T_i and T_j are. Currently, the only knowledge we have is that it must be the case that 0^{T_i} and 2^{T_j}.

Carrier sets are sets that contain all values of a certain type, and have the same name as that type. The distinction between a type and its carrier set is subtle. An observation can be in many different sets, but has exactly one type.

$$C =_{df} \{z^C \mid \mathbf{true}\}$$

The superscript C is the type and C is the carrier set. There is no ambiguity here, as in this metalanguage types are always superscript and only types can be superscript. The carrier set is useful in the following definition.

Schemas can have one of two forms. The basic form is the schema set, which is a schema with an unordered sequence of typed observations:

$$[\ldots z_i : C_i^{\mathbb{P}\, T_i} \ldots]$$

z_i is an observation, while C_i is a set with type $\mathbb{P}\, T_i$. The schema set is used in the other form, the atomic schema. This has the following structure, where S is a schema and P is a predicate.

$$[\, S \mid P\,]$$

An operator that we will be using frequently is the binding selection operator. This extracts the value of a given observation from a binding. For example,

$$\langle\, x \mapsto 0, y \mapsto 2\,\rangle.x = 0$$

More generally, this is defined as $\mapsto_0^=$:

$$\frac{}{\langle \ldots z_i \mapsto t_i \ldots \rangle . z_i = t_i} \mapsto_0^=$$

A simple schema without any propositions is defined as a set of bindings.

$$[\ldots z_i : C_i \ldots] =_{df} \{x \mid \ldots \wedge x.z_i \in C_i \wedge \ldots\}$$

Therefore, a given binding x exists in the defining set of bindings if, and only if, it has the following properties. Firstly, the names of every observation z_i in the schema are also in the binding x. Secondly, the value of observation z_i is in the set C_i. C_i may be a newly introduced set or, if it has the same name as a previously defined type, then it must be the carrier set of that type. Note that the schema type of x is same as the schema being defined, i.e. $x^{[\ldots z_i : C_i \ldots]}$. This means that the binding x only contains observations in the schema declarations. If x had another schema type, $x.z_i$ would not be valid for all z_i or x could include observations not equal to any observation z_i in the declarations of the schema. Consider the simple schema $[\, n : \{0, 1, 2\}\,]$. From the definition, $C_i = \{0, 1, 2\}$ and z_i can only be n. So, bindings in this schema have only one observation, n. Additionally, the value of n must be in $\{0, 1, 2\}$. The set of bindings is $\{\langle n \mapsto 0\rangle, \langle n \mapsto 1\rangle, \langle n \mapsto 2\rangle\}$.

When adding propositions P to schema S, the defining set of bindings need to satisfy P. The binding selection operator can be generalised over terms and over propositions, and $z.P$ is **true** if P is **true** when the variables from z are substituted with their binding values.

$$[\, S \mid P\,] =_{df} \{z \in S \mid z.P\}$$

The alphabet of a binding or schema is the set of observations it has. $\alpha[\ldots z_1 \ldots]$ is the set $\{\ldots z_1 \ldots\}$. $\preceq, \curlywedge, \curlyvee$ and $-$ are the subtype, intersection, union and subtraction operators for schema types. $\upharpoonright$ is the type restriction (or filtering) operation for bindings. $z \upharpoonright T$ restricts the binding z to the schema type T. For example, $\langle x \mapsto 0, y \mapsto 2\rangle \upharpoonright [x^{\mathbb{N}}] = \langle x \mapsto 0\rangle$. Note that the binding z must be an extension of T for this to be well-formed. In this example, the type of x in the binding must be $\mathbb{N}$. The main use of these operators is to ensure that our schema types are well-formed when performing more complex calculus. $z_0 \bigstar z_1$ is the binding concatenation operator, where z_0 and z_1 have non-overlapping type. For example, types T^{in} and $T^{out'}$ are types containing all before observations and types containing all after observations, respectively.

The last important type operator is priming. For example, $[\ldots z_1 \ldots]' = [\ldots z_1' \ldots]$. While we expect priming z gives us z', it is also true that $z'' = z$. This is not common in Z, but is useful in Z_C logic.

Next we present the terms of Z_C, variables, pairs, bindings, filtered bindings, free type

values and sets.

$$t^T ::= x^T \mid t^{[\ldots z^T \ldots]}.z \mid t^{T \times T_1}.1 \mid t^{T_0 \times T}.2$$
$$t^{T_0 \times T_1} ::= (t^{T_0}, t^{T_1})$$
$$t^{[\ldots z^T \ldots]} ::= \langle \ldots z \mapsto t^T \ldots \rangle$$
$$t^{T_0} ::= t^{T_1} \upharpoonright T_0 \text{ where } T_0 \preceq T_1$$
$$t^{\Upsilon} ::= c_i\ t^{\upsilon_{ij}}$$
$$t^{\mathbb{P}\,T} ::= \{z^T \mid P\}$$

These terms can then be used to build propositions.

$$P ::= \textbf{false} \mid t^T = t^T \mid t^T \in C^{\mathbb{P}\,T} \mid \neg\, P \mid P \vee P \mid \exists\, z^T \in C^{\mathbb{P}\,T} \bullet P$$

Finally, when we interpret the Z operations using the contractual interpretation (chaotic), we use the $\perp$ value. To include this value in our Z logic, we use $Z_C^{\perp}$. In $Z_C^{\perp}$ we add a new value $\perp^T$ to each type T in Z_C. Additionally, the carrier set of a type is defined differently in $Z_C^{\perp}$.

$$C =_{df} \{z^C \mid z \neq \perp\}$$

This means that the carrier sets do not include $\perp$. To convert a binding from Z_C to $Z_C^{\perp}$ (called lifting), we use the following definitions.

$$T_{\perp} =_{df}\ T \cup \{\perp\}$$
$$T^{*} =_{df}\ T_{\perp}^{in} \bigstar T_{\perp}^{out}$$

Additional definitions will be provided when and if necessary.

3.9 Summary

Z is used to write formal specifications, unambiguously describing what a system is meant to do. Strongly typed observations are declared in schemas to abstractly describe the state space and operations. We have looked at how operation schemas describe the change between before and afterstates. We have seen how we can use existential quantification to find the precondition of an operation and verify that we have a total or partial operation. Chaos and blocking interpretations were then introduced to cover what happens if an operation is used outside of its precondition. Finally, we looked at Z_C, the underlying logic of Z and how it defines the possible types and propositions that can be found in a Z specification.

Below, we give two more examples of using Z. These examples, along with the birthday book specification, will be used throughout the thesis and the majority of visualisations will be of these specifications.

3.10 Filling Jars

We have two jars, $j3$ has a volume of 3 pints, whereas $j5$ can contain 5 pints.

$$Jars ::= j3 \mid j5$$

$$\begin{array}{|l} max_fill : Jars \rightarrow \mathbb{N} \\ \hline max_fill = \{j3 \mapsto 3, j5 \mapsto 5\} \end{array}$$

Each jar currently contains a certain amount of liquid.

$$\begin{array}{|l} \textit{Level} \\ \hline level : Jars \rightarrow \mathbb{N} \\ \hline \forall j : Jars \bullet level(j) \leq max_fill(j) \\ \hline \end{array}$$

In the beginning, all jars are empty.

$$Init \mathrel{\widehat{=}} [\, Level \mid \operatorname{ran} level = \{0\} \,]$$

A (not full) jar can be filled completely.

$$\begin{array}{|l} \textit{Fill_Jar} \\ \hline \Delta Level \\ j? : Jars \\ \hline level(j?) < max_fill(j?) \\ level' = level \oplus \{j? \mapsto max_fill(j?)\} \\ \hline \end{array}$$

Or a (non-empty) one can be emptied completely.

$$\begin{array}{|l} \textit{Empty_Jar} \\ \hline \Delta Level \\ j? : Jars \\ \hline level(j?) > 0 \\ level' = level \oplus \{j? \mapsto 0\} \\ \hline \end{array}$$

Or we can pour the liquid of one jar into the other, until one is empty or full.

Transfer

$$\Delta Level$$
$$j1?, j2? : Jars$$
$$amount? : \mathbb{N}_1$$

$$j1? \neq j2?$$
$$amount? = min(\{level\, j1?, max_fill\, j2? - level\, j2?\})$$
$$level' = level \oplus \{j1? \mapsto level\, j1? - amount?, j2? \mapsto level\, j2? + amount?\}$$

3.11 Stopwatch Specification

The stopwatch has two observations. The current time value of the stopwatch and whether the stopwatch is paused or playing.

$$maxTime : \mathbb{N}$$

$$maxTime = 9999$$

Stopwatch

$$time : \mathbb{N}$$
$$playing : BOOL$$

$$time \leq maxTime$$

Initially the stopwatch is paused, and the time is 0.

Init

$$Stopwatch'$$

$$time' = 0$$
$$playing' = \textbf{false}$$

There is a button on the stopwatch that toggles between paused and playing.

Pause/Play

$$\Delta Stopwatch$$

$$time = time'$$
$$playing = \neg\ playing'$$

Resetting the stopwatch sets the time back to 0 and pauses the stopwatch.

Reset

$\Delta Stopwatch$

$time' = 0$

$playing' = \mathbf{false}$

Finally, each tick increases time if the stopwatch is playing. If the stopwatch is paused, time does not change. The behaviour when *maxTime* is reached has not yet been specified.

Tick

$\Delta Stopwatch$

$(playing = \mathbf{true} \land$

$time' = time + 1$

$\lor$

$playing = \mathbf{false} \land$

$time' = time)$

$playing = playing'$

Chapter 4

Refinement

Stepwise refinement, also called top-down design, is a way of creating computer programs [113]. Firstly, each of the functions of the system are described generally. Then more detail is gradually added until the functions are completely defined. This process has been formalised for specifications, and sets of rules have been created that allow us to prove that as changes are made the new specification still has the same underlying behaviour as the original specification.

In this chapter we introduce Z refinement. For Z, the refinement calculus provides a set of rules that are used to compare two specifications, the original abstract specification and a refined replacement concrete specification. This chapter only gives a brief introduction of the rules for Z refinement. Informally, a refinement is acceptable if it is impossible for an observer to notice the replacement. It is this principle that we use in our hypothesis; refinement can be used to characterise sound visualisations.

We cover two main types of refinement, operation refinement and data refinement. In operation refinement the concrete and abstract specifications have the same state space described by the same state schema. In data refinement, the state space of the abstract and concrete specifications can be different.

When using Z, there are three main abstraction techniques that are useful in the specification, but must be removed before implementation. These are partially defined operations, non-deterministic operations, and data structures that are not suitable for implementation.

The abstract specification is developed into a concrete specification with total, deterministic operations and an appropriate data structure. Z refinement is used to ensure that the concrete specification is consistent with the requirements of the abstract specification.

We hypothesise that Z refinement is suitable for characterising the soundness of specification visualisations. This is because a visualisation is similar to a concrete implementation for the purposes of soundness. The visualisation may use a data structure that is more suitable for visualising the behaviour of the specification and the type of visualisation we

are using may not allow non-deterministic or partial operations. Using refinement, we can test that the visualisation is consistent with the specification.

4.1 Principle of Substitutivity

How can we check that a refined specification still has the same behaviour as the original specification? One way is to consider it from the user's point of view, using the principle of substitutivity.

> *Principle of Substitutivity:* It is acceptable to replace one program by another, provided it is impossible for a user of the programs to observe that a substitution has taken place. If a program can be acceptably substituted by another, then the second program is said to be a *refinement* of the first. [29]

In this scenario, the user has learnt the behaviour of an abstract system. They interact with the system by using operations and inputting values, and can observe any output the system produces. This system is then replaced with a refined version. If a sequence of operations and input resulted in a particular output, then the user would expect to see the same output from the refined system. If the user cannot distinguish the difference between the two systems by using the same operations and inputs, then the principle of substitutivity is satisfied.

If the original system behaved nondeterministically, and could produce different outputs after the same sequence, then when using the same sequence in the refined system, the user would expect to see outputs that were in the set encountered previously. For example, a simple operation could output a natural number when used. If this operation was refined and the user observed the output being negative, they could then tell that the behaviour of the operation has changed. If, however, the refined operation produces natural numbers less than a thousand, or indeed, less than ten, then the user would not be able to definitively tell the difference between the two systems. They may suspect that a change has taken place, however the nature of nondeterminism means it is entirely possible for the system to produce a particularly long chain of small numbers.

Another way of refining the system in a way the user cannot distinguish is when a refined operation can be used in more states. This is because the user can only test the refined system by using operations and inputs that were possible in the abstract system. For example, our birthday book specification only allows the user to remove a name from the book if the name is *known*. We can refine this operation by outputting an error message when the user attempts to remove an invalid name. The principle of substitutivity says that this is a valid substitution, as the user cannot detect the substitution by only removing

names that are known. They would need to remove a name that is not known, but this is not possible in the abstract specification.

4.2 Preconditions and Postconditions

The precondition of an operation is a set of states and inputs for which the behaviour of the operation has been specified. Trying to use the operation outside of its precondition will result in unspecified behaviour. For example, an operation that removes an item from a set may have a precondition that requires the item is in the set to start with. The postcondition of an operation is a set of states that may be the result of an operation being used. For example, an operation that removes an item from a set may have a postcondition that the item is no longer in the set.

Let's consider the strength of the preconditions and postconditions. Proposition A is stronger than proposition B if $A \Rightarrow B$. For example, because $x = 1 \Rightarrow (x = 1 \lor x = 2)$, $x = 1$ is the stronger proposition. Strengthening $(x = 1 \lor x = 2)$ to $x = 1$ means what used to work in states where x is either 1 or 2 now works only in states where $x = 1$, i.e. fewer states. A stronger precondition means the operation is defined in fewer states, while a stronger postcondition means that there are fewer possible afterstates, or that the operation is less nondeterministic. A precondition that is **true** is the weakest precondition, and the operation is defined in all states. The strongest condition is **false** because an operation with **false** as its precondition is unusable in all states, and an operation with a **false** as its postcondition has no valid afterstates.

By comparing the preconditions and postconditions of the operations in the system with their refined versions, a set of rules has been created that lets us prove that no behaviour has been lost. These rules allow us to weaken the preconditions and strengthen the postconditions of the operations. Weakening the precondition means that the operation is now defined in a larger state space, while the behaviour in previously defined states should not be changed. Strengthening the postcondition means removing nondeterminism by removing elements from the set of possible afterstates. We cannot add new elements to this set, nor can we remove every element. As an example, imagine we have an operation that simply increases a number if the number is not greater than 10. We can weaken the precondition by adding the condition that if the number is greater than 10 then it is unchanged when the operation is used. Additionally, we can reduce the nondeterminism (strengthen the postcondition) by incrementing the number by one each time.

4.3 Operation Refinement (Derrick and Boiten)

Here we introduce a set of operation refinement rules from John Derrick and Eerke A. Boiten [29]. These rules compare two operation schemas, one from the abstract specification, AOp, and another from the concrete specification, COp. These operations have the same input and output observations, and operate over the same state space $State$.

To show that a concrete specification is an acceptable substitute, and that a refinement relation exists between the abstract and concrete specifications, we test that each operation in the concrete specification has two properties: applicability and correctness.

Applicability requires that the concrete operation is defined everywhere the abstract operation was defined. If the operation is no longer defined in a state when it was defined in the abstract operation then the user will realise the operation has been changed when trying to use the operation in this state. Applicability allows us to weaken the precondition of the operation.

Correctness requires that the concrete operation should map into the range of the abstract operation everywhere the abstract operation is defined. This means that the concrete operation cannot change the behaviour of the abstract operation by resulting in a different afterstate when the operation is used. Correctness allows us to strengthen the postcondition of the operation.

These conditions can be formalised in the following way. An operation COp is an operation refinement of an operation AOp over the state space $State$ with the same set of inputs $?AOp$ and set of outputs $!AOp$ if and only if:

Applicability property:

$$\forall State;\ ?AOp \bullet \mathbf{pre}\, AOp \Rightarrow \mathbf{pre}\, COp$$

Correctness property:

$$\forall State;\ State';\ ?AOp;\ !AOp \bullet \mathbf{pre}\, AOp \land COp \Rightarrow AOp$$

Note that the correctness property only requires that $COp \Rightarrow AOp$ where the operation has been defined. This is because we are using the standard contractual (chaotic) interpretation, and we do not need to enforce the correctness property on the chaotic behaviour outside the precondition. If using the blocking interpretation, the correctness property can be simplified.

$$\forall State;\ State';\ ?AOp;\ !AOp \bullet COp \Rightarrow AOp$$

If these rules are satisfied by COp and AOp, then COp satisfies the requirements of AOp. Additionally, COp can have reduced non-determinism and be specified in a wider set of states compared to AOp.

Consider the following example. We have a state schema with a single observation of a natural number.

$$State \mathrel{\widehat=} [\, n : \mathbb{N} \,]$$

We have an abstract operation that increases n. How much we increase n by is not important at this stage, so we use nondeterminism to abstract away this unimportant detail.

$$\begin{array}{|l} \multicolumn{1}{l}{AIncrease} \\ \hline \Delta State \\ \hline n' > n \\ \hline \end{array}$$

As the system is developed further we need to decide exactly how much n is increased by. We propose that the following operation schema that increases n by one is a valid substitution.

$$\begin{array}{|l} \multicolumn{1}{l}{CIncrease} \\ \hline \Delta State \\ \hline n' = n + 1 \\ \hline \end{array}$$

Now, we show that the underlying behaviour of $AIncrease$ has not been changed by showing that the refinement rules hold. We start by finding the preconditions of each of the operations.

$$\mathbf{pre}\, AIncrease = \mathbf{true}$$
$$\mathbf{pre}\, CIncrease = \mathbf{true}$$

Both operations are total so showing the applicability property holds is trivial.

$$\forall\, State;\ ?AOp \bullet \mathbf{pre}\, AIncrease \Rightarrow \mathbf{pre}\, CIncrease$$

$\equiv$ (Substitution)

$$\forall\, n : \mathbb{N} \bullet \mathbf{true} \Rightarrow \mathbf{true}$$

$\equiv$ ($\mathbf{true} \Rightarrow \mathbf{true}$)

$$\mathbf{true}$$

However, we are removing nondeterminism. The concrete operation $CIncrease$ is correct if it implies $AIncrease$.

$$\forall\, State;\ State';\ ?AOp;\ !AOp \bullet \mathbf{pre}\, AIncrease \land CIncrease \Rightarrow AIncrease$$

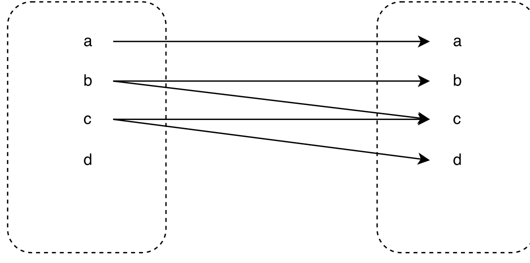


Figure 4.1: An operation shown as a partial relation

≡(Substitution)

$$\forall\, n, n' : \mathbb{N} \bullet \textbf{true} \land n' = n + 1 \Rightarrow n' > n$$

≡ ($n' = n + 1$ or not)

$$\forall\, n, n' : \mathbb{N} \bullet n' = n + 1 \Rightarrow n' = n + 1 \land n' > n \lor$$
$$n' \neq n + 1 \land n' = n + 1 \Rightarrow n' \neq n + 1 \land n' > n$$

≡(Simplify)

$$\forall\, n, n' : \mathbb{N} \bullet n' = n + 1 \Rightarrow n' = n + 1 \land \textbf{true} \lor$$
$$\textbf{false} \Rightarrow n' \neq n + 1 \land n' > n$$

≡ ($\textbf{false} \Rightarrow P, Q \Rightarrow Q$)

$$\forall\, n, n' : \mathbb{N} \bullet \textbf{true} \lor \textbf{true}$$

≡

true

Because both rules are **true** this is a valid refinement. Using the same example, if we instead decided to change the constraint to $n' = n - 1$ then this would not be a valid refinement.

4.4 Woodcock Operation Refinement

Rather than conceptualising refinement in two parts, weakening preconditions and strengthening postconditions, we can instead consider refinement as simply reducing nondeterminism as described by Woodcock and Davies [114]. This approach uses total operations. If either the abstract or concrete operations are only partially specified, then we instead use their *totalised* versions.

Figure 4.1 shows a partial relation that we will be using as an example. First the operations are lifted and totalised. Lifting means we are including $\perp$ as a possible state. This state is understood to mean undefined, erroneous, and generally out of our control.

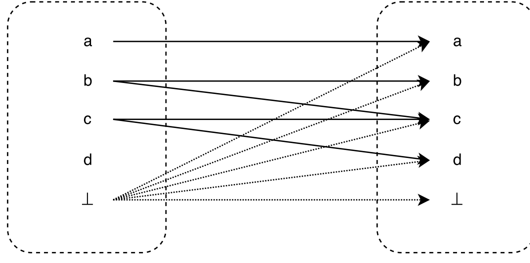


Figure 4.2: Lifting the relation

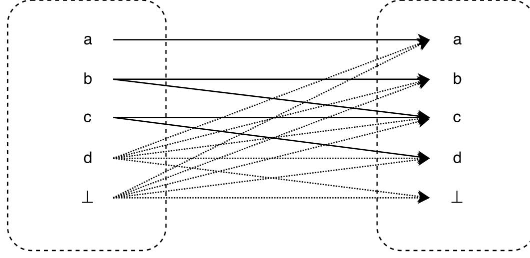


Figure 4.3: Totalising the relation

This is a state we do not want our system to be in, however using an undefined operation may result in this state. Figure 4.2 shows how $\bot$ is included, and how it maps to all afterstates. Totalisation means that we are expanding the states in the state space in which the operation is defined to include all states. The afterstates of the previously unspecified states can be any state, including $\bot$. In Figure 4.3, d was previously outside the domain of the relation, and so now maps to all afterstates after totalisation.

We write $\overset{\bullet}{\rho}$ to denote the totalised form of ρ. The lifted totalisation of a set of bindings has been defined as follows:

$$\overset{\bullet}{U} =_{df} \{z_0 \bigstar z_1' \in T^* \mid \mathit{Pre}\ U\ z_0 \Rightarrow z_0 \bigstar z_1' \in U\}$$

So, binding $z_0 \bigstar z_1'$ is included in $\overset{\bullet}{U}$ in two ways. Firstly, if the binding was part of the original operation U. Secondly, if the binding was not in U, then it is included only if z_0, the beforestate of the binding, does not satisfy the precondition of U. The before and afterstate of the binding may be $\bot$, because $z_0 \bigstar z_1' \in T^*$. So, like the example shown in Figure 4.3, $\overset{\bullet}{U}$ adds bindings to U such that all states outside the precondition map to all possible afterstates, including $\bot$.

Once we have totalised our operations in this way it is now clear to see how they behave outside their previous preconditions. There is also a simple rule for when a specification is a refinement of another specification: If the lifted totalised concrete operation is a subset of the lifted totalised abstract operation, then it is a valid refinement.

$$U_C \sqsubseteq_{W\bullet} U_A =_{df} \overset{\bullet}{U_C} \subseteq \overset{\bullet}{U_A}$$

For example, Figure 4.4 is a subset of the example in Figure 4.3, because although some mappings have been removed, all mappings in Figure 4.4 are present in Figure 4.3. Woodcock

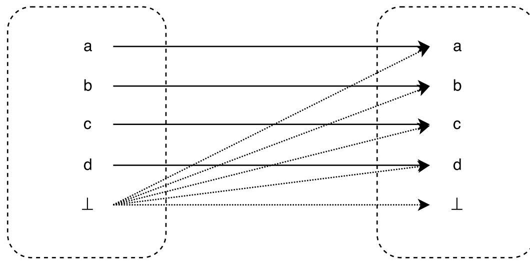


Figure 4.4: Refinement of the relation

refinement tests if nondeterminism is reduced in the totalised forms of the operations. Each beforestate that had multiple possible afterstates *including* d now only has one possible afterstate. Although this definition only tests the reduction of nondeterminism, when we consider the untotalised versions of the operations, we can see that nondeterminism has been reduced for b and c and the precondition has been weakened to include d. The same underlying principles apply to both of the refinement rules shown here.

4.5 Data Refinement

In addition to just refining the operations, it may also be useful to refine the underlying data structure or state space. For example, we may want to change the type of an observation from a set to a list. This is particularly useful when changing from the abstract data types used in early specifications to more concrete data types used in the later specifications or executable programs. Data refinement is similar to operation refinement, but the abstract state space and concrete state space can be different. We will be using data refinement in this thesis as the visualisations will often have a different representation of the state space.

The details of how and why data refinement works will not be presented here, instead we refer to existing literature [29]. However, the main principles of substitution, weakening preconditions and strengthening postconditions are still the same. We will be using the most common way of checking data refinement, downwards simulation.

Downwards simulations use a retrieve relation R that relates the abstract state space with the concrete state space. The retrieve relation is written as a schema that includes the abstract and concrete state schemas. The choice of predicate depends on how the state spaces are related.

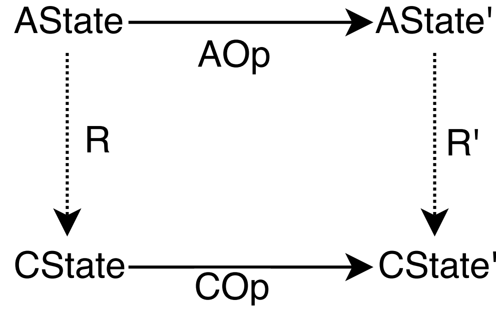


Figure 4.5: Refinement using downward simulation

The following example shows how we might use a retrieve relation. We have decided to change the state space of the Jars example such that instead of using a partial function we use simple numbers to represent how full each jar is. $CLevel \mathrel{\widehat=} [j1, j2 : \mathbb{N} \mid j1 \leq 3 \wedge j2 \leq 5]$ We can then use R to relate the two different state spaces. How much liquid is in the first jar is now specified as $j1$ instead of $level(j3)$.

$$
\begin{array}{l}
R \\
\hline
Level \\
CLevel \\
\hline
level(j3) = j1 \\
level(j5) = j2
\end{array}
$$

Data refinement extends operation refinement by considering the changes in the state space and how the system is initialised. Data refinement also includes properties for applicability and correctness, and while the underlying idea is unchanged, the retrieve relation R has been included. R lets us compare the abstract and concrete operations despite the fact that they have different state spaces.

In Figure 4.5 we can see an abstract operation change the abstract state and a concrete operation change the concrete state. These are then connected by R. If we start in the topleft state and follow the arrows we can see that application of relation R followed by the operation COp can be matched by the operation AOp followed by R'.

This is used to change the operation refinement properties. Previously the correctness property was written as:

$$\forall State;\ State';\ ?AOp_i;\ !AOp_i \bullet \mathbf{pre}\, AOp_i \wedge COp_i \Rightarrow AOp_i$$

Correctness still requires that the concrete operation should map into the range of the abstract operation everywhere the abstract operation is defined. The difference is that COp will now result in a concrete state. So, the range of AOp is changed to the concrete state space using R. Based on the observation above, we now apply R before the operation COp_i

and follow AOp_i with R'. The $\exists AState'$ is used to hide any abstract afterstate observations.

$$\forall AState;\ CState;\ CState';\ ?AOp_i;\ !AOp_i \bullet \mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow$$
$$\exists AState' \bullet AOp_i \land R'$$

Similarly to operation refinement, there exists a simpler correctness rule when using the blocking interpretation.

$$\forall AState;\ CState;\ CState';\ ?AOp_i;\ !AOp_i \bullet R \land COp_i \Rightarrow \exists AState' \bullet R' \land AOp_i$$

Applicability still requires that the concrete operation is defined everywhere the abstract operation was defined. Previously this was written $\forall State;\ ?AOp \bullet \mathbf{pre}\, AOp \Rightarrow \mathbf{pre}\, COp$. However, the concrete operation is now defined in the concrete state space. So we use R to simulate AOp being used in the concrete state space as well:

$$\forall AState;\ CState;\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$$

Finally, we have a new property that must hold for a valid refinement relation to exist. The system is being initialised in a different state space and we need to ensure that we don't start the system in a completely different state. $CInit$ should start the system in the range of $AInit$ and we use R simulate $AInit$ starting in the concrete state space rather than the abstract state space. Again, $\exists AState'$ is used to remove abstract observations from the right hand side of the implication. Initialisation property:

$$\forall CState' \bullet CInit \Rightarrow \exists AState' \bullet AInit \land R'$$

Let us try using applying data refinement to our simple jars example. We have refined our Init operation into $CInit \mathrel{\widehat=} [CLevel' \mid j1' = 0 \land j2' = 0]$.

$$\forall CLevel' \bullet CInit \Rightarrow \exists Level' \bullet AInit \land R'$$

$\equiv$(Substitution)

$$\forall CLevel' \bullet j1' = 0 \land j2' = 0 \Rightarrow \exists level' \bullet \operatorname{ran} level' = \{0\} \land$$
$$level'(j3) = j1' \land level'(j5) = j2'$$

$\equiv$(One-point Rule)

$$\forall CLevel' \bullet j1' = 0 \land j2' = 0 \Rightarrow$$
$$0 = j1' \land 0 = j2'$$

$\equiv$

true

Next we will check applicability and correctness hold for a refined Fill operation.

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{CFill}} \\ \hline \Delta CLevel \\ j? : JARS \\ \hline j? = j3 \land j1 < 3 \land j1' = 3 \land j2' = j2 \lor \\ j? = j5 \land j2 < 5 \land j2' = 5 \land j1' = j1 \\ \hline \end{array}$$

The precondition of $Fill_Jar$ is $[Level;\ j? : Jars \mid level(j?) < max_fill(j?)]$ while the precondition of $CFill$ is $[CLevel;\ j? : Jars \mid j? = j3 \land j1 < 3 \lor j? = j5 \land j2 < 5]$.

$$\forall AState;\ CState;\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$$

$$\forall Level;\ CLevel;\ j? \bullet level(j?) < max_fill(j?) \land level(j3) = j1 \land level(j5) = j2$$
$$\Rightarrow j? = j3 \land j1 < 3 \lor j? = j5 \land j2 < 5$$

$\equiv$(Substitution)

$$\forall Level;\ CLevel;\ j? \bullet j? = j3 \land level(j3) < 3 \lor j? = j5 \land$$
$$level(j5) < 5 \land level(j3) = j1 \land level(j5) = j2 \Rightarrow$$
$$j? = j3 \land j1 < 3 \lor j? = j5 \land j2 < 5$$

$\equiv$(Substitution)

$$\forall Level;\ CLevel;\ j? \bullet j? = j3 \land j1 < 3 \lor j? = j5 \land j2 < 5 \land level(j3) = j1 \land level(j5) = j2$$
$$\Rightarrow j? = j3 \land j1 < 3 \lor j? = j5 \land j2 < 5$$

$\equiv$

true

So the applicability property holds, next we look at the correctness property.

$$\forall AState;\ CState;\ CState';\ ?AOp_i;\ !AOp_i \bullet \mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow$$
$$\exists AState' \bullet AOp_i \land R'$$

$$\forall Level;\ CLevel;\ CLevel';\ j? : Jars \bullet level(j?) < max_fill(j?) \land level(j3) = j1 \land$$
$$level(j5) = j2 \land (j? = j3 \land j1 < 3 \land j1' = 3 \land j2' = j2 \lor$$
$$j? = j5 \land j2 < 5 \land j2' = 5 \land j1' = j1) \Rightarrow$$
$$\exists Level' \bullet level(j?) < max_fill(j?) \land level' = level \oplus \{j? \mapsto max_fill(j?)\}$$
$$\land level'(j3) = j1' \land level'(j5) = j2'$$

$\equiv$(One-point Rule)

$$\forall \mathit{Level};\ \mathit{CLevel};\ \mathit{CLevel}';\ j? : \mathit{Jars} \bullet \mathit{level}(j?) < \mathit{max_fill}(j?) \land \mathit{level}(j3) = j1 \land$$
$$\mathit{level}(j5) = j2 \land (j? = j3 \land j1 < 3 \land j1' = 3 \land j2' = j2 \lor$$
$$j? = j5 \land j2 < 5 \land j2' = 5 \land j1' = j1) \Rightarrow \mathit{level}(j?) < \mathit{max_fill}(j?) \land$$
$$(\mathit{level} \oplus \{j? \mapsto \mathit{max_fill}(j?)\})(j3) = j1' \land$$
$$(\mathit{level} \oplus \{j? \mapsto \mathit{max_fill}(j?)\})(j5) = j2'$$

$\equiv$(Simplify $j?$)

$$\forall \mathit{Level};\ \mathit{CLevel};\ \mathit{CLevel}';\ j? : \mathit{Jars} \bullet \mathit{level}(j?) < \mathit{max_fill}(j?) \land \mathit{level}(j3) = j1 \land$$
$$\mathit{level}(j5) = j2 \land (j? = j3 \land j1 < 3 \land j1' = 3 \land j2' = j2 \lor$$
$$j? = j5 \land j2 < 5 \land j2' = 5 \land j1' = j1) \Rightarrow \mathit{level}(j?) < \mathit{max_fill}(j?) \land$$
$$(j? = j3 \land (\mathit{level} \oplus \{j3 \mapsto 3\})(j3) = j1' \lor$$
$$j? = j5 \land (\mathit{level} \oplus \{j5 \mapsto 5\})(j3) = j1') \land$$
$$(j? = j3 \land (\mathit{level} \oplus \{j3 \mapsto 3\})(j5) = j2' \lor$$
$$j? = j5 \land (\mathit{level} \oplus \{j5 \mapsto 5\})(j5) = j2')$$

$\equiv$(Simplify $\oplus$)

$$\forall \mathit{Level};\ \mathit{CLevel};\ \mathit{CLevel}';\ j? : \mathit{Jars} \bullet \mathit{level}(j?) < \mathit{max_fill}(j?) \land \mathit{level}(j3) = j1 \land$$
$$\mathit{level}(j5) = j2 \land (j? = j3 \land j1 < 3 \land j1' = 3 \land j2' = j2 \lor$$
$$j? = j5 \land j2 < 5 \land j2' = 5 \land j1' = j1) \Rightarrow \mathit{level}(j?) < \mathit{max_fill}(j?) \land$$
$$(j? = j3 \land 3 = j1' \lor j? = j5 \land \mathit{level}(j3) = j1') \land$$
$$(j? = j3 \land \mathit{level}(j5) = j2' \lor j? = j5 \land 5 = j2')$$

$\equiv$(Rearrange)

$$\forall \mathit{Level};\ \mathit{CLevel};\ \mathit{CLevel}';\ j? : \mathit{Jars} \bullet \mathit{level}(j?) < \mathit{max_fill}(j?) \land \mathit{level}(j3) = j1 \land$$
$$\mathit{level}(j5) = j2 \land (j? = j3 \land j1 < 3 \land j1' = 3 \land j2' = \mathit{level}(j5) \lor$$
$$j? = j5 \land j2 < 5 \land j2' = 5 \land j1' = \mathit{level}(j3)) \Rightarrow \mathit{level}(j?) < \mathit{max_fill}(j?) \land$$
$$(j? = j3 \land \mathit{level}(j3) < 3 \land j1' = 3 \land j2' = \mathit{level}(j5) \lor$$
$$j? = j5 \land \mathit{level}(j5) < 5 \land j2' = 5 \land j1' = \mathit{level}(j3))$$

$\equiv$

true

So we have found a valid refinement relation despite the fact that the state spaces of the abstract and concrete specifications are different.

The refinement calculi we have looked at so far have required two sets of matching operations, for example *AIncrease* and *CIncrease*. When using refinement each abstract operation AOp_i is tested with its matching concrete operation COp_i. Operation pairs need to have the same set of input and output observations $?AOp_i$ and $!AOp_i$.

Methods of refinement have been developed that allow us to relax these constraints. The input and output can be changed [15], and operations can be split apart [30]. Additionally, the refinement calculi discussed here are not restricted to only Z specifications, and many languages have custom refinement rules that are specific to that language [57, 77]. In this thesis we will be focusing on Z refinement, particularly operation refinement and downward simulation data refinement.

4.6 Summary

In this chapter we have introduced three types of refinement. We will be using all three throughout the thesis. Refinement allows us to compare two specifications and verify that the preconditions have not been strengthened and nondeterminism has not increased. This allows us to remove abstraction or use a different state space without changing the underlying behaviour. If an operation is enabled in the abstract specification it must also be enabled in the concrete specification. If a transition between two states is possible in the concrete specification it must also be possible in the abstract specification.

Chapter 5

Visualisations

We are interested in the soundness of specification visualisations. Visualisations are tools for helping users understand and analyse complex information. In this section we discuss who uses specification visualisations and why. Then, by building on the ideas in the discussion, we explain soundness and why it is an important visualisation property.

5.1 Visualisation Users

We begin by identifying the three main user groups for these visualisations. Visualisations can be used to verify and validate specifications. Different users will require different types of visualisations to help assist completing different tasks.

Firstly, there is the creator(s) of the specification. The specifier has been given an informal specification of the system, and their goal is to create a formal specification that accurately models the client's requirements. This could be either the initial specification, or a specification that is intended to be a refinement of the original specification. If they are working in a team, then they may begin by only specifying a small part of the overall system, and then later integrating this with the larger system. Visualisations help these users identify errors created during the writing of specifications and help with the comparison of formal and informal specifications. The state space of these visualisations can be reduced by incorporating mathematical language and observations from the specification.

Clients also make use of specification visualisations. They are not expected to have any knowledge of the formal language of the specification, however they still need to understand the specification to help ensure it matches with their informal requirements. They will primarily be using the visualisation to help validate the specification by identifying any differences between the formal specification and their vision. Walk-throughs and simulations of the system being used are helpful for this task. Domain-specific visualisations can help lower the semantic distance by showing the system using graphical language that the client

can understand.

The third user group consists of programmers. They are given the formal specification and their goal is to build a functioning program that matches the specification. A visualisation of the specification gives an overview of what needs to be implemented. While they may also have a copy of the client's original informal specification, the formal specification the programmer receives will have been heavily refined and may include many details that are not defined in the original specification. Visualising these details can save the programmer time and energy when parsing the specification.

5.2 Visualising Z Specifications

A Z specification specifies what a system does. A visualisation of a specification should provide a clearer and more understandable way of showing the same information.

The Z specification contains operation schemas, and these operation schemas describe how the state of the system changes when the operation is used, depending on the current state and input. The specification itself has no restrictions on what states are reachable, as each of the operations are independent from one another and the behaviour of each operation can be specified for any possible state, even if it is not possible to reach the state when looking at the system as a whole. Therefore, if a simple visualisation is made to show the behaviour of the whole specification, it needs to show how the operations change the state. More sophisticated visualisations can also show additional emergent behaviour that is not explicitly defined in the specification. This could include showing how the operations interact to create a working system, or which operations need to be applied in what order to reach particular states. However, the bare minimum we require is for the visualisation to show how the operations change the state of the system.

5.3 Visualisation Soundness

If the visualisation is accurate and does not mislead the user, then the visualisation is sound. We assess this by firstly, requiring that if the visualisation shows some behaviour, such as an operation changing the state, it must also be possible in the specification. Secondly, we require that all operations enabled in a particular state of the specification must also be enabled in that state of the visualisation. Our goal is to formally characterise this property.

Both the client and the specifier rely on the visualisation being sound. For the client, if the visualisation is unsound, then they do not know if the specification that has been created matches their informal requirements. An unsound visualisation could make an incorrect specification appear to be correct. For the specifier, an unsound visualisation could hide bugs in the specification or visualise bugs that do not exist. The programmer does

not rely as heavily on the soundness of the visualisation, as they were only using the visualisation to get an idea of the system. If there is a discrepancy between the visualisation and the specification it could cause some confusion, but only the specification needs to be implemented to successfully build a correct program.

Although visualisations are intended to be clear and usable, often the visualisation may be large, confusing, or written using a graphical language the user does not understand. We separate these problems from the problem of soundness. To this end, we simply assume that no matter how large, complex, and confusing the visualisation is the user can still understand any visualisation perfectly. Of course this is not realistically possible, however these assumptions vastly simplify the process of characterising soundness as we no longer need to consider the limitations of individual users.

There are two ways we can formalise the user's understanding of a visualisation. These are based on two opposing epistemological paradigms [10]: that knowledge is objective or that knowledge is relative. The first method is to use the formal semantics of the graphical language, where they exist. The semantics of the visualisation are assumed to be the objective truth that we can use consistently and without ambiguity.

Alternatively, relativism assumes that there is no objective truth and instead different viewpoints each have their own truth. Different people using the visualisation can interpret it differently based on their world view and their understanding of the specification and graphical language. A formal model can be created that represents the user's understanding of the visualisation. If the meaning of the visualisation is clear and unambiguous then this formalisation should be the same for all users. However it is possible that two people can interpret the same image differently or the same person might interpret the visualisation differently over time. Using this philosophy means that when a sound visualisation is looked at from a different viewpoint it may be unsound. This paradigm implies that we cannot formally characterise the soundness of visualisations.

We could limit our investigation to only include visualisations with formal semantics. However, many useful visualisations are informal. Because of this we do not ignore the problems that relativism introduces. Instead, our method allows different viewpoints to be formalised. This means that we can now assume that the visualisation does not need to have a single objective truth and that different people can interpret it differently.

The purpose of our specification visualisations is to provide a better way to understand and analyse the behaviour of the specifications. We use the semantics of the visualisation or the formalisation of the user's understanding of the visualisation.

5.3.1 Soundness and Completeness

In logic soundness has a distinct meaning. A deductive system is sound if and only if all provable things are 'true' with regards to the semantics. Testing for the property of logical soundness is different from what we are investigating, however we will briefly discuss it. A method for visualising specifications has the soundness property if every visualisation that is creates is 'true' with regards to the specification. What does it mean for a visualisation to be 'true' with regards to the specification? In fact, this is our main research question written in a different form, and part of why we have chosen to use the term 'sound' when describing visualisations.

In logic soundness is often discussed alongside completeness, as both properties deal with how the proof system and semantics work together [17]. If every 'true' formula can be proven then the system is complete. A method for visualising specifications has the completeness property if it can create every 'true' visualisation. It is not necessarily useful to prove that such a method is complete for the following reasons. Firstly, this does not mean that every visualisation that is created will be helpful or even usable. A method that can create awful visualisations of anything would still be complete, though perhaps not sound. Secondly, different types of visualisations can be more or less suitable for a particular specification. Therefore, different methods are used to visualise different specifications and it is not necessary to be able to use one particular method to visualise any specification.

5.4 ProZ

ProZ is a model checker and simulation tool that is used to analyse and visualise Z specifications [89]. We are using ProZ as an example of a tool that can create animated and static visualisations. Figure 5.1 shows ProZ with the birthday book example opened. Two main uses of ProZ that we will be discussing are the simulation and graphical animation tools, although we will also briefly discuss the state diagram visualisation that ProZ can display.

ProZ is an extension of ProB, which is a tool for another specification language, B [67]. The extension works by converting Z specifications into B specifications behind the scenes, although no understanding of B is needed to use ProZ. Z specifications, written in LaTeX, can be opened directly using ProZ, without needing to be converted into a proprietary file type. This means we can quickly analyse our existing specifications. Z specifications also include informal descriptions of the schemas, and this does not need to be removed when opening the specification.

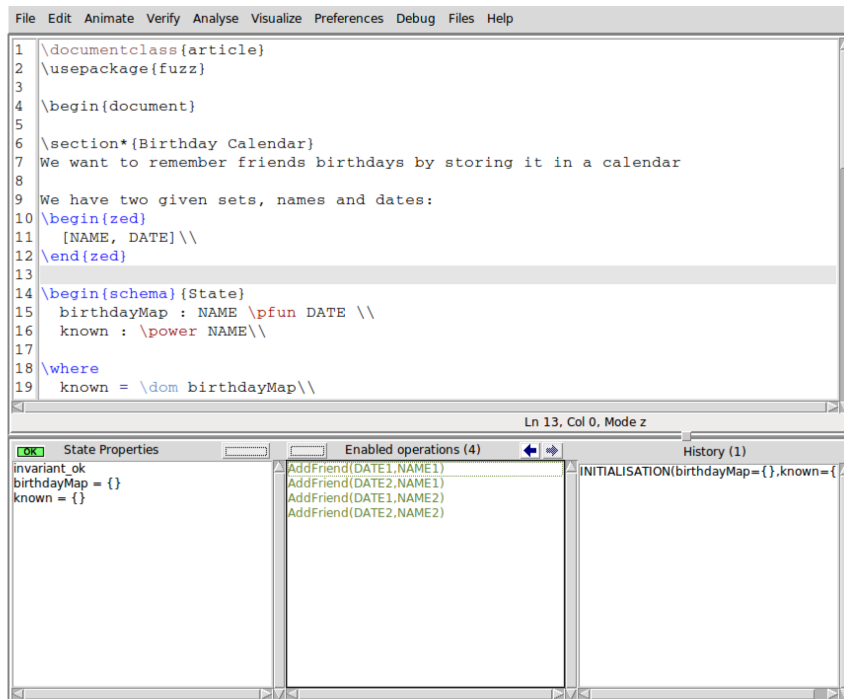


Figure 5.1: ProZ

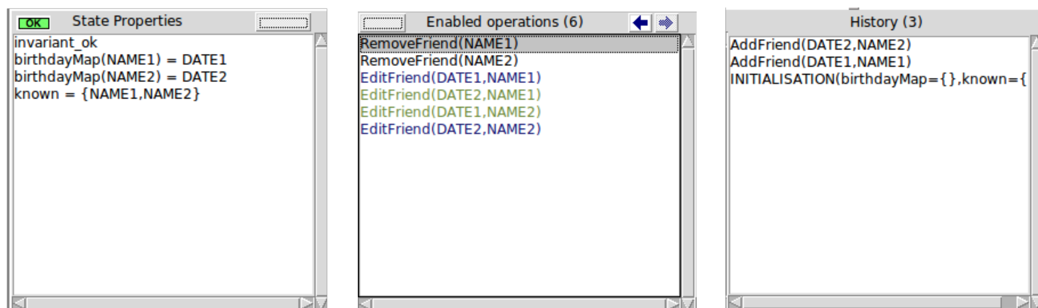


Figure 5.2: State Properties

Figure 5.3: Enabled Operations

Figure 5.4: History

ProZ Simulation

The first tool we will discuss is the simulator. When we introduced specification languages, we mentioned that they cannot be executed as a typical programming language program could be. However, by introducing some restrictions, the ProZ simulation tool allows us to 'run' the specification, simulating initialising, providing inputs, applying operations and viewing the output. Basically, it searches for observation values that satisfy the constraints of the schemas.

The values of the observations for the current state of the simulation can be observed. Figure 5.2 shows the state properties after two names have been added to the birthday book. Being able to visualise the values of observations before and after using operations is useful for any user.

At each step of the simulation, ProZ provides a list of operations that are enabled in the current state. This is shown in Figure 5.3. The user can pick any of these, then the

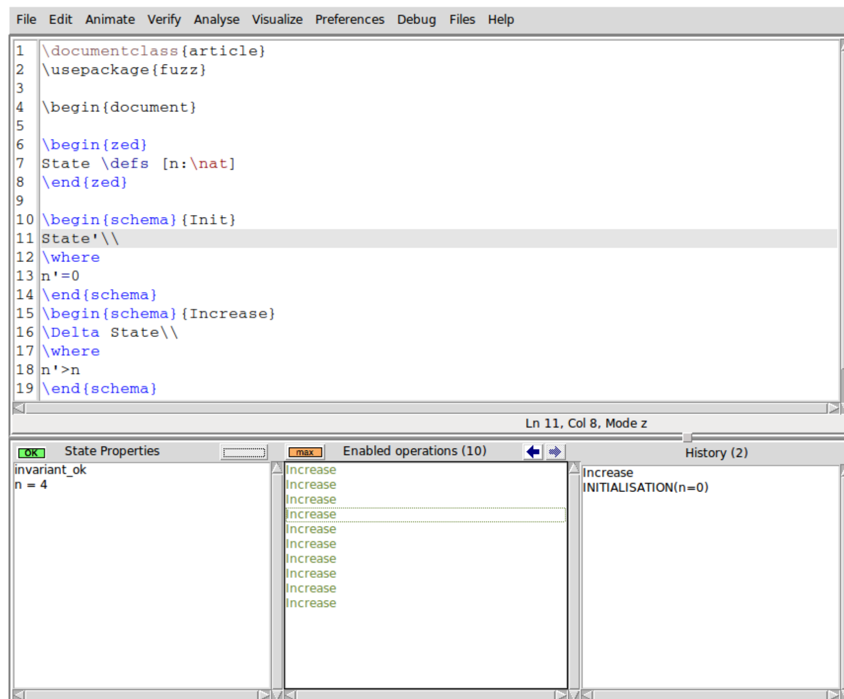


Figure 5.5: Increase Specification

current state values are updated and a new list of operations are provided. This allows the user to simulate the specification being run. Each operation in the list is a combination of multiple properties. First is the name of the operation being used, second is the value of the inputs being provided, third is any resulting output that will occur. So, for example, if an operation can be used in the current state with two valid inputs, it will appear twice in the list. The operations are colour coded depending on the resulting state, with different colours for entering an unexplored state, a previously explored state or if the state does not change.

Figure 5.4 shows the history panel of ProZ. Here, the user can see the operations that have been used so far in the simulation. When clicked, the simulation reverts to a previous state, so that the user does not need to restart the simulation to explore multiple branches.

It is important to discuss the limitations of the ProZ simulation. The ProZ simulation does not have unlimited memory. This is an obvious limitation of any physical program, and any computer program we try to use to analyse our specification will be restricted by this. One limitation is the size of sets. Z specifications can use observations which have types that have infinitely large carrier sets. This could be as simple as $x : \mathbb{N}$, or we can use a given type like $name? : NAMES$ where $[NAMES]$ is understood to be the set of all names. In Figure 5.3 we see that we can only edit friends to $DATE1$ or $DATE2$. Additionally, we cannot use the *AddFriend* operation in this state. This is because the size of the $NAME$ and $DATE$ carrier sets is restricted by the tool setting, in this case, to 2. In the simulation, $DATE1$ and $DATE2$ are the only values with type $DATE$.

Nondeterminism is handled by explicitly providing the user with the option to choose a particular operation. If an operation can nondeterministically enter two different states, then it will appear in the list twice, and the user can pick which state to enter. Because the user can choose the resulting state, it no longer behaves nondeterministically, and if the user did not understand this limitation they may believe that the specification itself is deterministic. Additionally, the number of operations provided to the user is limited. If there are too many possible afterstates, inputs or outputs, then not all possible combinations will be provided for the user to simulate. This could cause a user to believe that a particular operation is not possible since it was not shown in the list of possible operations. However, a warning is provided if ProZ was not able to show every possibility. In Figure 5.5, an orange box labelled *max* is shown next to the enabled operations title. This is because *Increase* is nondeterministic, so the operation appears in the list multiple times for each possible afterstate. However, the number of operations shown to the user is limited to 10, so although the specification says that n can be increased to any natural number, we can only choose from 10 options. Finally, the size of *maxInt* has been limited to 20, so at most this simulation could only increase n to 20.

The ProZ simulation is blocking. If the current state is not in the precondition of an operation, then the user does not have the option to use it. This is fine for specifications that are intended to be blocking, however if the specification uses the chaotic interpretation then a user may expect to be able to use an operation outside its precondition.

So, with all these limitations, can this simulation be used to understand the specification without being misled? Is the ProZ simulation unsound? We will discuss this further in chapter 7.

5.5 Graphical Animation

ProZ allows us to define an *animation function*[69, 70]. By adding images and the following schemas to the jars specification, ProZ can display the grid of images seen in Figure 5.6 while the specification is being simulated. This provides the user with a graphical animation that updates each time an operation is used.

Having the maximum fill of all jars helps us in the visualisation:

$$global_maximum : \mathbb{N}$$

$$global_maximum = max(\mathrm{ran}(max_fill))$$

Firstly, we declare a type for the images where the names of the elements refer to the file names of the GIF files for each of the images.

$$Images ::= Filled \mid Empty \mid Void$$

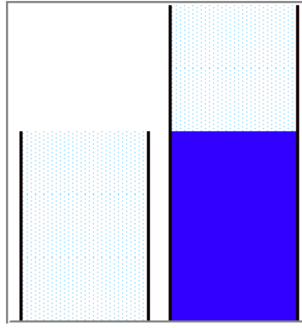


Figure 5.6: 2x5 Grid of Images

There are several simple naming requirements that must be met to use this graphical animation. The animation function, named *animation_function*, must be defined in a schema called *ProZ_Settings*. The animation consists of a grid of images that is updated in each new state. The *animation_function* maps a coordinate to an image where $(1 \mapsto 1)$ is the upper-left corner. *animation_function_default* is a helpful optional addition that is used to create a background for the animation. In this example the background is simply white.

ProZ_Settings

$$
\begin{array}{l}
\mathit{Level} \\
\mathit{animation_function_default} : (\mathbb{N} \times \mathit{Jars}) \pfun \mathit{Images} \\
\mathit{animation_function} : (\mathbb{N} \times \mathit{Jars}) \pfun \mathit{Images} \\
\hline
\mathit{animation_function_default} = (1 \mathinner{\ldotp\ldotp} \mathit{global_maximum} \times \mathit{Jars}) \times \{\mathit{Void}\} \\
\mathit{animation_function} = \\
\quad (\{l : 1 \mathinner{\ldotp\ldotp} \mathit{global_maximum};\ c : \mathit{Jars} \mid l \leq \mathit{max_fill}\ c \bullet \\
\qquad \mathit{global_maximum} + 1 - l \mapsto c\} \times \{\mathit{Empty}\}) \oplus \\
\quad (\{l : 1 \mathinner{\ldotp\ldotp} \mathit{global_maximum};\ c : \mathit{Jars} \mid l \leq \mathit{level}\ c \bullet \\
\qquad \mathit{global_maximum} + 1 - l \mapsto c\} \times \{\mathit{Filled}\})
\end{array}
$$

The graphical animation allows the user to show the simulation to clients with an even lower barrier of entry. Because the graphical animation is limited to a grid of images, it does not have the power of a custom graphical animated visualisation. However it is often sufficient, and will serve as a good comparison when we discuss animated visualisations later.

In Figure 5.6, our y coordinate is 1 up to the global maximum of our jars and our x coordinate each of the jars. We then build the visualisation in layers. Firstly the white background is created. As a set, this looks like $\{(1, j3, \mathit{Void}), (1, j5, \mathit{Void}), (2, j3, \mathit{Void}), (2, j5, \mathit{Void}), ...\}$. This is then overridden by empty jar images. If the y coordinate is less than the maximum level of that jar we draw an empty jar. The variable l here is used to draw up from the bottom of the grid so that we don't end up with upside down jars. Finally, this is overridden again by filled jar images. If we are between the bottom of the jar and the current fill level, then we

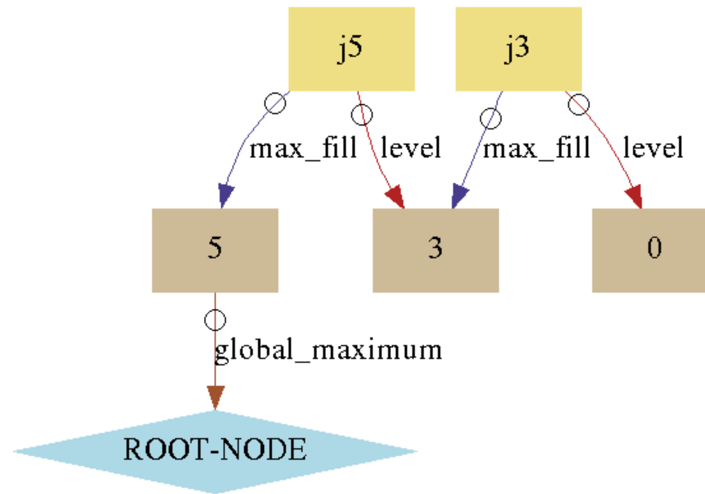


Figure 5.7: Graph of State in Jars

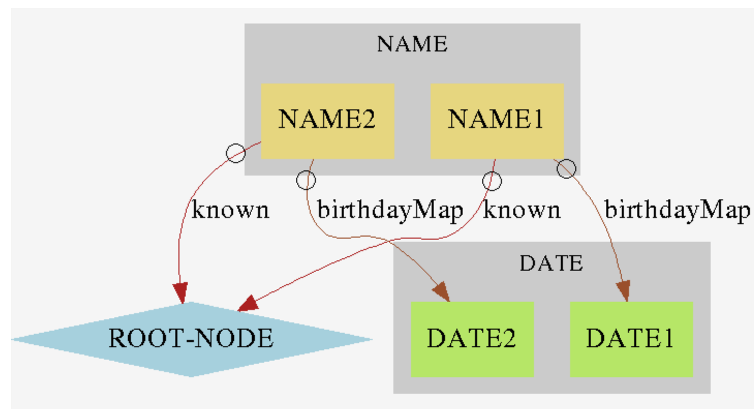


Figure 5.8: Graph of State in Birthday Book

draw a filled jar. This gives us a set like $\{(1, j3, \mathit{Void}), (1, j5, \mathit{Empty}), ..., (5, j3, \mathit{Empty}), (5, j5, \mathit{Filled})\}$ depending on the current value of *level*. This visualisation is then drawn and updated by ProB during simulation.

Other ProZ Visualisations

ProZ can also be used to display the specification as a state graph, and can do this in a few different ways, including visualisations of the state, operations, and invariants.

The figures 5.7 and 5.8 demonstrate another way ProZ can display states of the specification. The states shown are the same as figures 5.6 and 5.2. These visualisations show values of sets such as $5, j5$, and $NAME2$ as coloured squares in a graph. These nodes are then connected based on the value of the observation in the state being visualised. So, if the current level of $j5$ is 3 or $DATE2$ is the birthday of $NAME2$ according to *birthdayMap* then these links are created and appropriately labelled. However, not all observations can be broken down into pairs of values. For example, $global_maximum = 5$. In this case our visu-

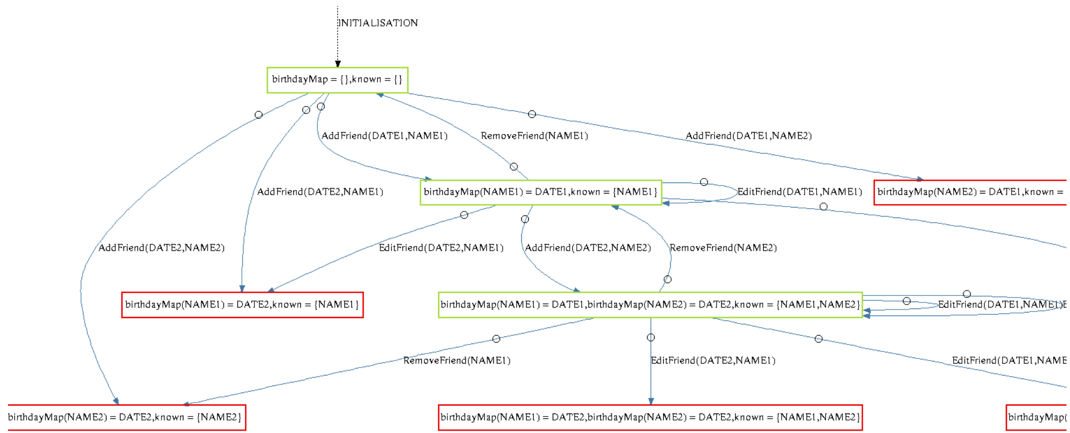


Figure 5.9: Graph of Explored States in Birthday Book

alisation links the 5 node with "*ROOT-NODE*". This node is included in the visualisation when the visualisation requires a valueless node to connect to.

We will mostly be looking at the state diagram visualisation of the currently explored state space. This state diagram is connected to the simulation, and shows the states that have been explored, the current state, and unexplored states that have been seen as possible afterstates but have not yet been visited. In Figure 5.9 we show the graph that ProZ can generate after adding two names to the birthday book.

5.6 Conclusion

Visualisations are a useful validation tool. They are helpful for many types of users and are particularly helpful for users that would have difficulty understanding specifications otherwise. We can visualise the same specification in many different ways. This allows us to approach problems from multiple angles and can reveal hidden information.

In his thesis A.J. Pretorius concludes:

> It was shown that visualization can assist in understanding the complex system behavior represented by state transition graphs. On the one hand, by supporting explorative visual analysis, it was shown that users are enabled to gain a better intuitive understanding of their data. On the other hand, by supporting focused analysis, visualizations of transition graphs enable users to investigate particular features and answer specific questions about their data. Moreover, communication between users themselves and between users and other stakeholders was substantially enhanced by giving visual form to inherently abstract data. [92] (p.133)

We are focusing on specification visualisations and this allows us to narrow the scope of

what defines a visualisation and what its purpose can be. We further narrow the scope by only focusing on visualisation soundness while ignoring features like aesthetics and usability.

There exists a fundamental gap between the informal and the formal. We discussed two ways to help bridge this gap while formalising visualisations.

Finally, we introduced the ProZ tool and displayed some of the specification visualisations it can create. These examples helped highlight some of the difficulties of creating sound visualisations.

Chapter 6

State Diagrams

This chapter covers the first type of visualisation we will be investigating in this thesis: state diagrams. We have chosen state diagrams as the main visualisation type as they are a commonly used visualisation method for state and operation-based designs. Z specifications are typically written with state and operation schemas, so state diagrams are an appropriate type of visualisation.

There are many variations of state diagrams: they can use different graphical notations. For example, labels on states and transitions can have different meanings and appearances. Because we are visualising the system in different ways, we will be looking at several different types of state diagram. We will then present different formalisations of these state diagrams that we can later use to prove that the visualisations are sound.

The classic form of a state diagram is a directed graph with states, transitions, input symbols, and an initial state. This has been used to visualise finite-state machines. A state diagram has the elements $(Q, \Sigma, Z, \delta, q_0)$:

- Q is a finite set of states, normally drawn as circles labelled with symbols or words.
- Σ is a finite set of input symbols.
- Z is a finite set of output symbols.
- δ is a finite set of transitions, normally drawn as arrows connecting the beforestate to the afterstate, and labelled with an input and output symbol.
- q_0 is the start state of the state diagram, and will be drawn as either an arrow with no origin pointing to the state, or a state drawn with two circles.

In the classic form, the sets are finite, and each transition is labelled with a single input. However, we are using the state diagrams to visualise Z specifications. In Z, sets can be infinite in size, and the operations can accept multiple input observations and have output observations. Additionally, the *Init* schema can be nondeterministic and also have input

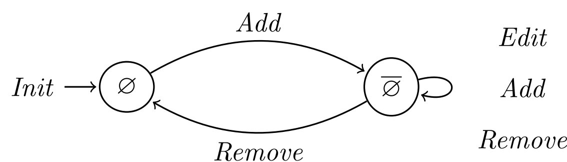


Figure 6.1: Empty or not empty

observations, so the state diagram can have multiple initial states. This means that the classical state diagram definition will only be suitable for the simplest Z visualisations.

Because of this we present our own variations of state diagrams. We make minor changes to the state diagram definition that allows us to create a greater variety of visualisations. This includes state diagrams with no initial state, labels with symbols that match the specification observations, graphical syntactic sugar, and more. Many variations of state diagrams exist and we could choose to use existing semantics instead of creating our own. However, the type of state diagram chosen is not important. We choose to include a larger variety of handmade state diagrams rather than a few particular types.

The examples that are presented in this section have been chosen as representatives of their subtype of state diagram visualisation. The discussion of what is actually being visualised, and whether this choice of content or level of abstraction is useful to a potential user, is not relevant here. Typically, we assume that each of these visualisations could have been created to communicate some or all of the specification behaviour to a potential user.

The first state diagram we look at is a visualisation of the Birthday Book. It has two states, empty and not empty, and shows that when we add a name to our empty birthday book it is no longer empty. Then, we can add more names, edit names and remove names, and removing a name may result in the book being empty again.

Figure 6.1 is a simple visualisation that shows the three operations being used. The input of the operations is not shown, so we assume that the choice of input does not change the behaviour of the state diagram, since whichever name we add to the book when it is empty will result in it becoming not empty.

$$Q = \{\varnothing, \overline{\varnothing}\}$$
$$\Sigma = \{Add, Edit, Remove\}$$
$$Z = \{\}$$
$$\delta = \{(\varnothing, Add) \mapsto \overline{\varnothing}, (\overline{\varnothing}, Edit) \mapsto \overline{\varnothing}, (\overline{\varnothing}, Add) \mapsto \overline{\varnothing}, (\overline{\varnothing}, Remove) \mapsto \overline{\varnothing}, (\overline{\varnothing}, Remove) \mapsto \varnothing\}$$
$$q_0 = \varnothing$$

This is also an example of an unsound visualisation. The reasons why are discussed in appendix B, however, a closer look reveals that after adding two names to the birthday book, removing one can cause the book to become empty.

6.1 State diagrams with no current state

The following example focuses on state diagrams that do not have a 'current' state. If we consider a deterministic finite-state machine, then the system is always in exactly one state at any given time, and this current state may change when a transition occurs. If we instead consider a trace of operations, then as we execute this trace the current state changes. However, Z specifications typically cannot be executed, and the definition of Z we provided in chapter 3 did not include a definition of a 'current' state that would change as the specification is used. Instead, an operation simply specifies possible afterstates for all possible beforestates where the precondition is **true**.

The following specification is provided to highlight the point that state diagrams do not need to always be in exactly one state at a given time, and additionally, that states do not need to be reachable to have specified behaviour. The operation schema *NOperation* is total, and so the afterstate is specified for all possible beforestates.

$NState \mathrel{\widehat=} [\, n : \mathbb{N} \,]$

This is also shown in Figure 6.2. This example is unusual for a state diagram because in the classic form of a state diagram, a transition has a beforestate, input, and an afterstate. The next state of the diagram depends on the input symbols on the outgoing transitions of the current state.

This visualisation does not have a current state. However, it can still be used to visualise the before and afterstates of *NOperation*. The value of n is visualised with labels on the states. $n = 0$ satisfies the condition for both of the top states, and this visualisation shows that when $n < 2$ the afterstate will be $n = 0$. Additionally, the state space is disjoint, as states are unreachable from other states. When state diagrams are used to visualise state machines the state space would not be disjoint as if a state is unreachable in the state machine as it would not be drawn in the state diagram. However, we choose to visualise the behaviour of the specification even in "unreachable" states.

A definition for this type of state diagram needs to include the propositions on the labels and the observations in the state space being visualised, in this example, n. This state diagram can be defined using the triple (Q, S, δ), where:

- Q is a set of states, drawn as circles and labelled with a proposition.

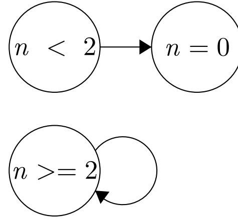


Figure 6.2: Visualisation of NOperation

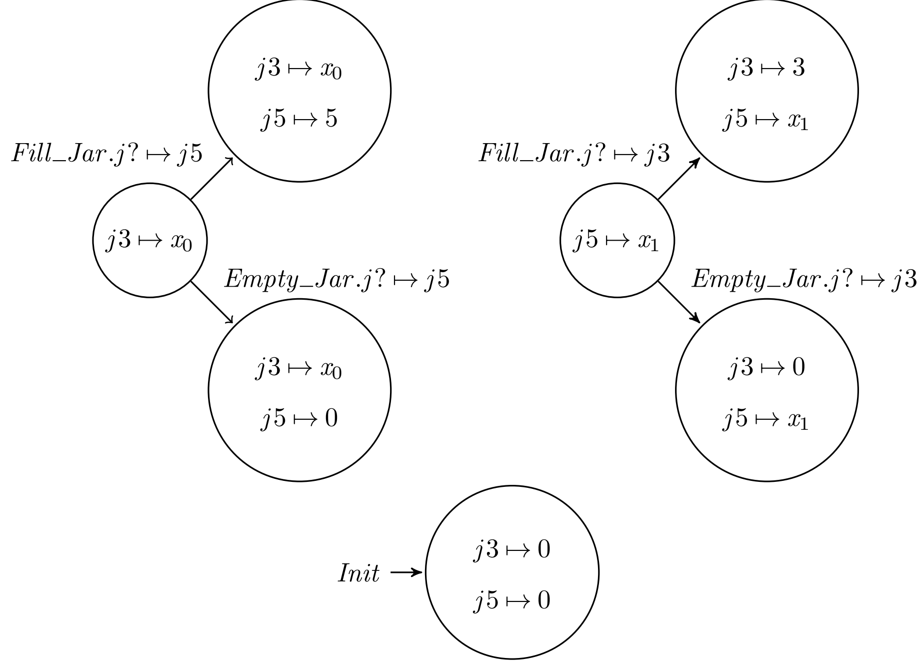


Figure 6.3: State Diagram of Jars

- S is a set of observations.
- δ is a set of transitions, drawn as arrows connecting the beforestate to the afterstate.

Because there is only one operation being visualised, and there is no *Init* schema, many of the elements of the state diagram definition have been removed. For the state diagram in Figure 6.2, there is one observation, n, and three states; $\{(n < 2), (n = 0), (n \geq 2)\}$. There are two transitions in δ; $\delta(n < 2) = (n = 0)$ and $\delta(n \geq 2) = (n \geq 2)$. This is the full definition of this visualisation.

Next, Figure 6.3 shows how the *Empty_Jar* and *Fill_Jar* operations of the jars specification work. The fill and empty jar operations change the current amount of liquid of $j3$ or $j5$ to the most it can hold or zero, and do not affect the amount of liquid in the other jar. Like the previous example, this state diagram is not required to have exactly one current state at all times.

There is a lot of information in the labels of this visualisation. First, we have labels on the states themselves, which refer to the observations $j3$ and $j5$. Additionally, we have values x_0 and x_1, which are used to make sure that the jar not being filled does not change.

If the amount in the jar before using the operation was x_0, then it must be x_0 afterwards. Finally, the transitions are also labelled. Here, we give the name of the operation, the input observation name and its value.

Rather than the propositions, as in the previous example, we are using bindings as labels on the states. When we want to visualise an operation being used, we do not trace through the state diagram from the initial state until we have reached what we need. Instead, we look for the operation we are interested in, and we can quickly see how the state will change based on the beforestate of the relevant transition(s). For example, we want to know how the *Fill_Jar* operation works. There are two *Fill_Jar* transitions, and we see that if the input observation $j?$ is $j5$, then the value of $j5$ becomes 5 and the value of $j3$ does not change. We can see similar behaviour when we fill $j3$.

This type of state diagram has a different definition that uses bindings. We use the tuple (Q, Σ, δ, q_0), however the elements of the tuple have different meanings:

- Q is a set of states, where each state is labelled with a binding.
- Σ is a set of operations, which can have input and output observations.
- δ is a set of bindings, normally drawn as arrows connecting the beforestate to the afterstate, and labelled with the operation being used and the value of the input and output observations.
- q_0 is the initial state.

q_0 is the start state of the state diagram, and is drawn as an arrow with no origin pointing to the state. For example, q_0 can be written as the binding $\langle j3 \mapsto 0, j5 \mapsto 0 \rangle$.

The bindings in δ are the transitions. The values of the observations in the bindings are found in the before and afterstates and the transition, as well as the transition label which determines the input and output observation values. For example, a simple transition binding could be written as $\langle j3 \mapsto 0, j5 \mapsto 0, j? \mapsto j3, j3' \mapsto 3, j5' \mapsto 0 \rangle$.

However, the state diagram in Figure 6.3 still does not fit this definition, because of the states labelled $j3 \mapsto x_0$ and $j5 \mapsto x_1$. x_0 and x_1 have not been included in this definition. Additionally, these two states do not include both observations, so the schema types of the transitions are not well-formed. To fix this, we observe that each transition represents multiple bindings. For example, $\langle j3 \mapsto 0, j5 \mapsto 0, j? \mapsto j3, j3' \mapsto 3, j5' \mapsto 0 \rangle$ and $\langle j3 \mapsto 1, j5 \mapsto 1, j? \mapsto j3, j3' \mapsto 3, j5' \mapsto 1 \rangle$ are part of the same transition. For the full formalisation of this visualisation, and for the proof that it is sound, see appendix C.

This is a useful way to visualise Z specifications as it is more abstract than a typical state diagram that has a current state. Additionally, the labels on the states of the state diagram are a powerful tool that is used here to both identify the beforestate values of observations,

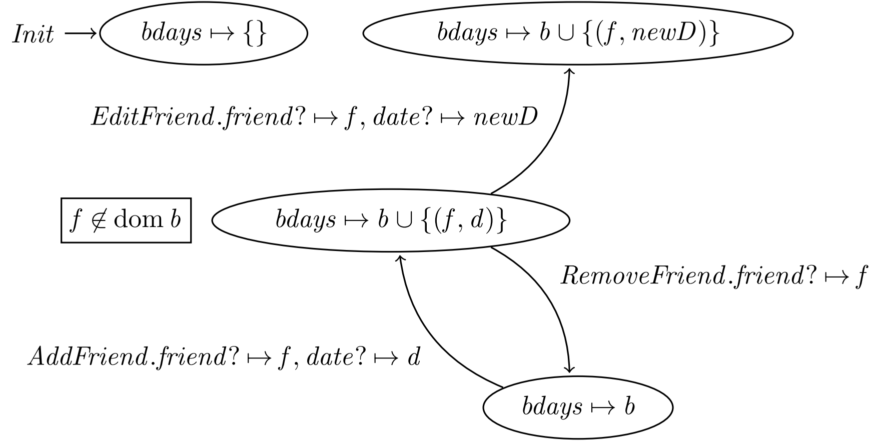


Figure 6.4: State diagram of birthday book

and use these values to calculate the afterstate values. However, this type of state diagram does not behave intuitively like a user may expect if they are accustomed to state machines.

A similar type of visualisation is shown in Figure 6.4 to visualise the birthday book example. Unlike the previous example, this state diagram includes a box that adds a constraint to f. This state diagram includes particularly complex labels which combine the propositions and bindings we saw in the previous two examples. This complexity allows us to visualise the entire birthday book specification with only four states and three transitions. These transitions show that if our book contains friend f with birthday d, we can edit the date to a new date $newD$, or remove friend f from the book. These operations do not affect the other names and dates in the book, b. Finally, we can add a given name to the book, provided that the friend is not already recorded in the book.

Each transition represents multiple bindings. There is no limit to the number of friends that can be added to the book, so b has infinitely many possible values. Let us look at two bindings in the *RemoveFriend* transition and match the observation values with the binding propositions in the diagram.

$$\langle bdays \mapsto \{(name1, date1), (name2, date2)\}, friend? \mapsto name1, bdays' \mapsto \{(name2, date2)\}\rangle$$
$$\langle bdays \mapsto \{(name1, date1), (name2, date2)\}, friend? \mapsto name2, bdays' \mapsto \{(name1, date1)\}\rangle$$

In the first binding, $b = \{(name2, date2)\}, f = name1$, and $d = date1$. The additional requirement that $f \notin \operatorname{dom} b$ is satisfied. In the second binding, $b = \{(name1, date1)\}, f = name2$, and $d = date2$.

In appendix D we look at bindings of the other operations: *EditFriend* and *AddFriend*. We find that the bindings of the visualisation transitions match the specification operations and therefore this visualisation is sound.

Figure 6.5: Birthday Book States where set size is two

6.1.1 ProZ State Space

Not all visualisations need to be handmade. Drawing custom visualisations lets us be more informal, explore particular parts of the specification, choose the layout, and other aesthetics that can make a visualisation more usable. However, automatically generated visualisations have their own strengths, particularly speed of generation and accuracy of information. Both styles are useful and each serves a different objective. We discussed the Z visualisation tool ProZ in section 5.4, and show examples of the state diagrams it can be used to generate below.

Visualising the entire state space, where each state represents unique values for the observations, is a difficult problem. Even simple specifications can have an infinite number of states, and even finite state spaces may still be far too large to draw or read comfortably.

The visualisations in Figure 6.5 and Figure 6.6 have been generated using ProZ. The first has restricted the size of all sets and types to be two, while the second has been restricted to three. In Figure 6.6 the floating window shows the full state space, while the main window shows a zoomed-in portion. Because of the restrictions, only two or three names may be added to the birthday book, but this is still a large state space to explore. By comparing these generated visualisations with the previous visualisation we can see that the labels on the states include different information. The birthday book has two observations in the state schema, and both are faithfully included in the generated visualisation. However, the value of *known* is redundant information for the user, and was not included in the custom visualisation.

We do not need to generate the entire state space using ProZ (and in fact cannot due to limitations of memory). The visualisation in Figure 6.7 shows a particular state and the operations that can be used from this state. Note that because the size of the sets is still two for this example we have two outgoing operation transitions for *AddFriend* and *EditFriend*.

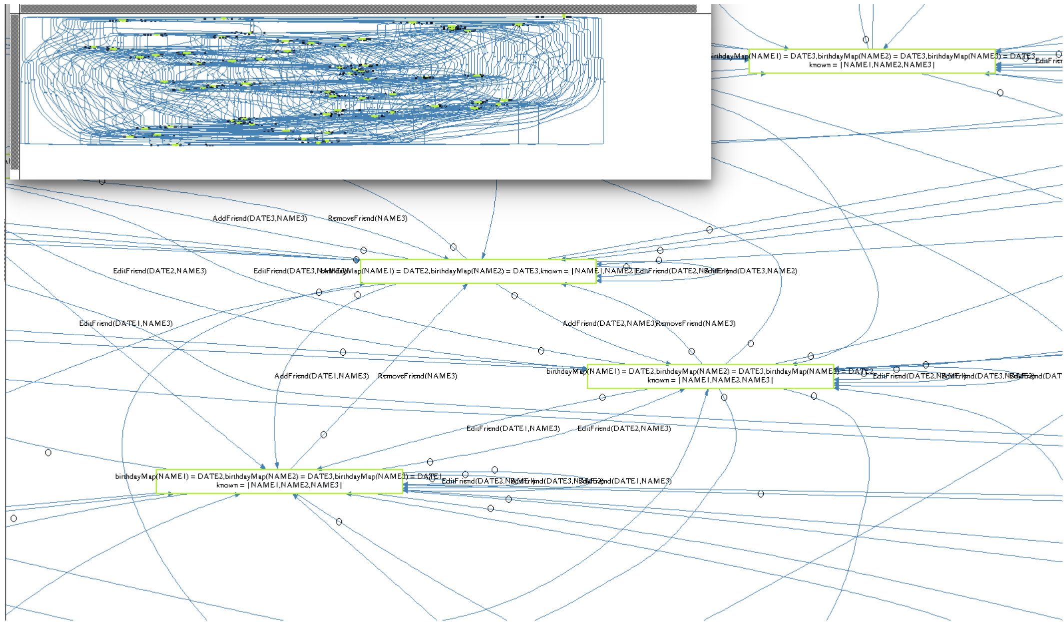


Figure 6.6: Birthday Book States where set size is three

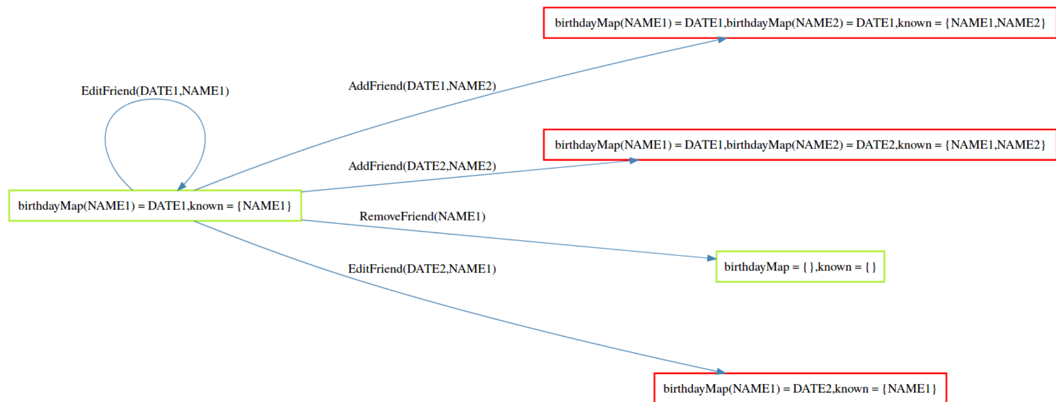


Figure 6.7: Visualisation of birthday book state

This sort of visualisation gives us an example of how each operation changes the state of the specification, given a beforestate and input values. Some states are more useful to visualise than others, for example the initial state, boundary states, and states where the precondition of operations are not satisfied. This lets us easily validate particular specification states. Additionally, if unexpected transitions are present or expected transitions are missing, it can also reveal errors in the specification. For example, in Figure 6.7 there is a selfloop on the state being visualised. Depending on the intentions of the client, it may be an error in the specification to be able to edit the date without changing it.

6.1.2 More State Diagram Types

The state diagram shown in Figure 6.8 is a visualisation of the stopwatch specification. The most notable property of this state diagram is the syntactic sugar used to reduce the

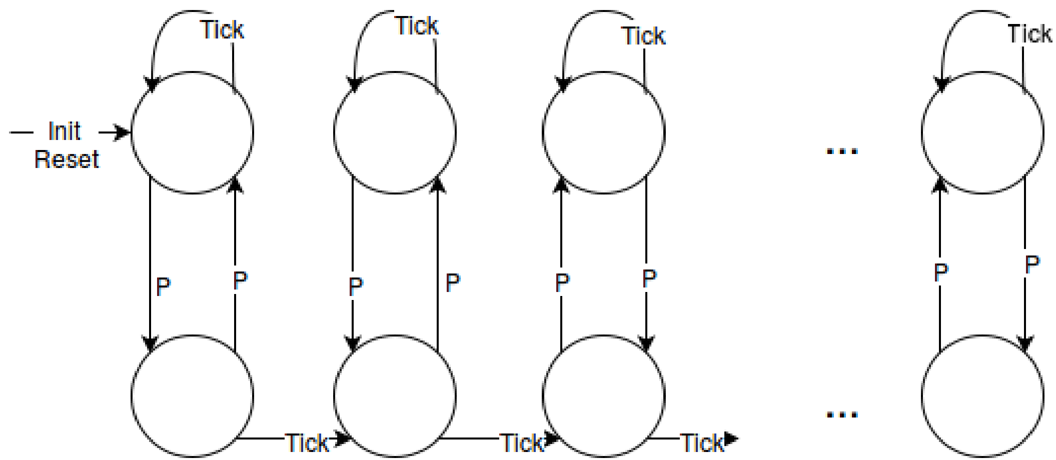


Figure 6.8: Visualisation of stopwatch

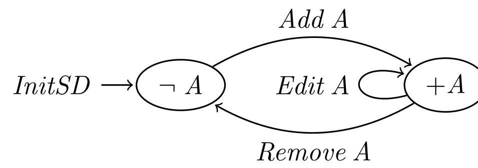


Figure 6.9: Add, Edit, and Remove A

number of states and transitions. Ellipses (. . .) are used to show a pattern continuing until a particular point, while the transition labelled *Reset* has no start state. This means that *Reset* can be used from any state, and will return the state diagram to the initial state. Another difference, compared to the previous visualisations we have presented so far, is that the states have no labels. The top row are states where the stopwatch is paused, and the bottom row has the stopwatch running. The time increases from left to right until we reach the maximum time allowed. Although this information could have been included as labels on the states, it has been excluded in this example as it is not needed for a user to understand the visualisation.

The state diagram in Figure 6.9 is similar to the earlier state diagram in Figure 6.1. Both are visualisations of the birthday book, have only two states, and the transitions are not labelled with all the input values we expect from the operations in the specification. This syntactic sugar allows us to simplify the visualisation when the value of the excluded observation is not relevant. In this example we are only visualising the operations that add, edit or remove person A from the book. The first state represents all books that do not contain A, while the second is all books that do contain A. The labels on the states here are just names to identify the states, and do not define the value of the *bday* observation, as we saw in Figure 6.4. Although *Add* and *Edit* have two input observations, only one has been provided in the label. So while $name? = A$, $date?$ is unconstrained and can have any value of type $DATE$. For example, when in state $\neg A$ we can add A with $DATE1$, $DATE2$, or

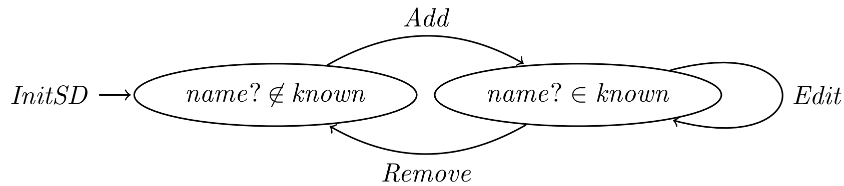


Figure 6.10: Precondition labels on states

any other, and the resulting state will be state $+A$, because the name A is now in the book.

This visualisation does not have transitions that show what happens if we try to add a different name to the book. If we assume that this state diagram is a sound and accurate visualisation of the birthday book specification then we should infer that an error has been found in the specification because a different name cannot be added to the birthday book. However, this diagram is only focused on visualising A and other names are outside its scope. Our main discussion about this type of *restricted* visualisation can be found in chapter 7.

In the birthday book example, each of the operations have the precondition that *name*? is in *known* or not in *known*. The state diagram in Figure 6.10 visualises this.

The observation names from the specification are included in the state labels. Rather than showing the values of observations or an informal name, these labels give the condition that allows the operation to be used, as well as demonstrating that the value of *known* has changed to satisfy the new condition of the afterstate.

In appendix E we use operation refinement to check if this visualisation is sound. Although this visualisation accurately shows the preconditions, the postconditions it shows are weaker than the specification operations. For example, *Remove* can remove every name in the birthday book at once. Therefore, this is not a sound visualisation.

To help reduce the size of the state space, we can merge similar states together. This can be seen in many of the previous examples. The state diagram in Figure 6.11 is another birthday book visualisation, where the entire state space has been merged into three states.

- StateT: birthday books that contain the pair (“Alan Turing” $\mapsto$ “June 23, 1912”)
- StateF: birthday books that contain the name “Alan Turing”, but with some different date.
- State0: birthday books that do not contain the name “Alan Turing”.

The labels on the transitions in the state diagram are different from previous examples. The operations are separated out into the cases where the name and date are equal to or different from “Alan Turing” and “June 23, 1912”, respectively. In the state diagram the operation names have been given subscript notes to represent the inputs in order to save space. For example, $Add_{A,T}$ is the add operation with inputs “Alan Turing” and “June 23, 1912”. $Edit_O$ is the edit operation where the name is not “Alan Turing” and the date is any

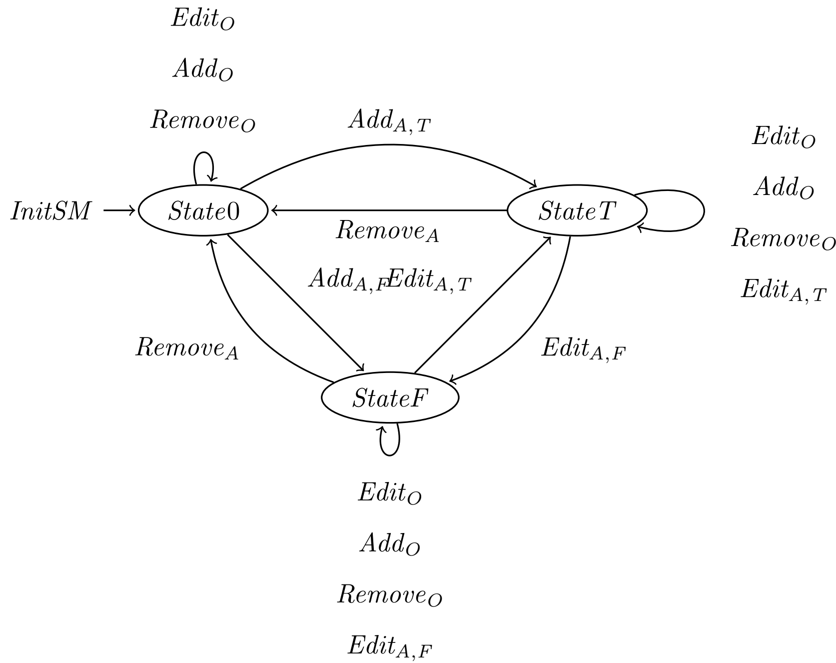


Figure 6.11: Alan Turing Visualisation

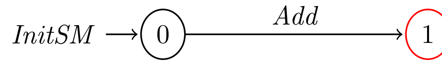


Figure 6.12: Small Visualisation

date. $Remove_{A,F}$ is the remove operation where the name is "Alan Turing" and the date is not "June 23, 1912" We formalise this visualisation using schema calculus in appendix F and use data refinement to show that this visualisation is sound.

The generated visualisations only showed part of the state space due to choice or restrictions on memory. In the Birthday Book visualisation shown in Figure 6.12 we make the same choice, showing that only *Add* can be used from the initial state. This may be all the information we want to validate, and so a small visualisation is all that is needed.

This introduces the idea of the *Unexplored* state, a state with no outgoing transitions. State diagrams that include unexplored states only visualise part of the specification, leaving the rest unexplored.

The state diagram in Figure 6.13 includes an unexplored state and has used colour to indicate the transitions that have been used and the states that have been explored. This lets us examine part of the birthday book specification, while being clear that it does not visualise the entire specification.

We have provided examples of several different types of state diagram visualisations. This covers only some of the many possible types, as the broad category of state diagram visualisations includes an enormous variety of different aesthetics and purposes. Furthermore, there is no universal formal semantics for these state diagram visualisations and in Section

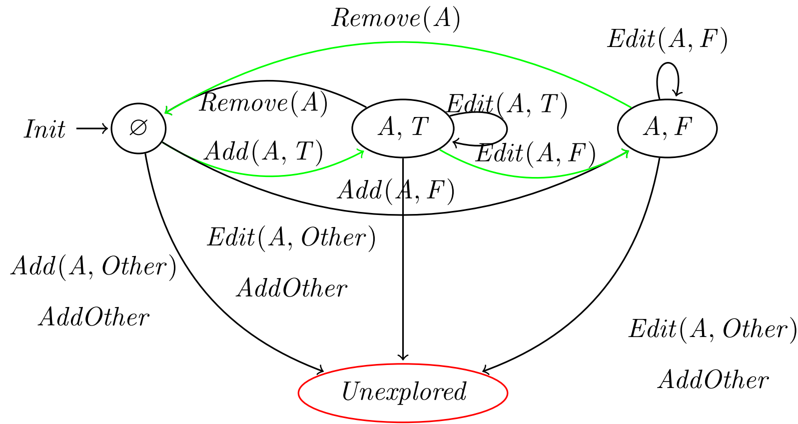


Figure 6.13: Unexplored Alan Turing Visualisation

5.3 we discussed how these visualisations can be interpreted differently by different people. Nevertheless, our hypothesis is that we can characterise the soundness of each of these types of visualisation, and more, by using refinement. To help narrow the scope, we will examine these examples looking for differences and similarities that we can use to present a consistent definition to be used in the rest of this thesis.

6.2 State Diagram Visualisation Definitions

Because we are only investigating the soundness of visualisations our definition of the visualisation does not need to include any aesthetic features such as colour, layout, or how exactly the labels are written. We have seen that the transitions on the labels have used many different characters including brackets, commas, $\mapsto$ and $=$. However, for all examples, the transitions show operations that can be used. Although the same meaning is implied, the labels on transitions varied in length and formality. $\mathit{Fill_Jar}.j? \mapsto j5$ provides the name of the operation being used with the input observation included as a mapping. To reduce the size of the labels, labels like $\mathit{Add}(\mathit{Alan}, T)$ used the context to imply that *Alan* was the value of *name*?, and T was the value of *date*?. When the exact input did not change the transition, the label could be simplified further, to *Add* or Add_O. The choice of label style should not change the underlying meaning of the transition. To meet these requirements, our definition will need to include two main properties: the operation being used, and the input observation values for that operation.

The states however have more substantial differences. For some examples the label simply helps identify the state, such as $\varnothing$ or $\overline{\varnothing}$. Other visualisations use the label as a conditional statement about the state observations, such as $j3 \mapsto x_0$. Some examples do not include state labels while others use colour to identify the special *unexplored* state.

Many of the state diagrams we show operate over the same state space as the specification. Others, such as Figure 6.3 operate over a different state space. State diagrams with

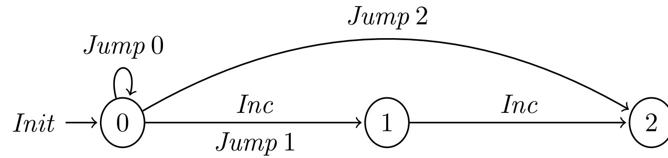


Figure 6.14: State Diagram of IncJump

unexplored states only visualise part of the state space of the specification. Diagrams that "merge" multiple specification states into a single visualisation state may be considered to have a different state space. However, Figure 6.10 only has two states, but references the *known* observation from the specification state schema. This shows that although the states have been merged, the underlying state space is the same as the specification and should therefore not be considered different.

Our goal is to find refinement relations between the visualisation and the specification. There are many different types of refinement and so we need to decide which refinement calculus we use. Rather than create a new custom refinement calculus we choose to use existing Z refinement as we are visualising Z specifications. By defining the visualisations in Z we can use Z refinement.

In an earlier section we introduced semantics for state diagrams based on the idea that states and transitions can be represented as bindings. To be well-formed we require that the states in the visualisation are not merged.

Our formalisation of the state diagram uses bindings, where each transition is represented as a binding and each operation in the state diagram is a set of bindings. Each transition has a beforestate, an afterstate and may have input and output values. A binding is built using these values. The bindings are collected into operations that can then be compared with the specification using refinement.

We present the state diagram (Q, Σ, δ, q_0) in Figure 6.14 as bindings. This is a visualisation of the simple *IncJump* specification:

$$\begin{aligned}
&\mathit{State} \mathrel{\widehat=} [n : \mathbb{N}] \\
&\mathit{Init} \mathrel{\widehat=} [\mathit{State}' \mid n' = 0] \\
&\mathit{Inc} \mathrel{\widehat=} [\Delta \mathit{State} \mid n' = n + 1] \\
&\mathit{Jump} \mathrel{\widehat=} [\Delta \mathit{State};\ n? : \mathbb{N} \mid n' = n?]
\end{aligned}$$

Firstly, the labels on the states refer to the value of n. This lets us easily build bindings matching each of the states.

$$Q = \{\langle n \mapsto 0\rangle, \langle n \mapsto 1\rangle, \langle n \mapsto 2\rangle\}$$

There are two operations in the *IncJump* specification, so $\Sigma = \{\mathit{Inc}, \mathit{Jump}\}$. We use the binding concatenation operation to build the transition bindings. For each transition we

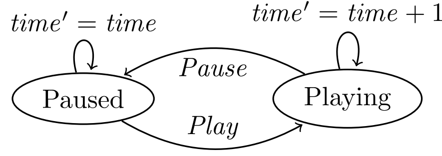


Figure 6.15: These diagram states each represent several specification states

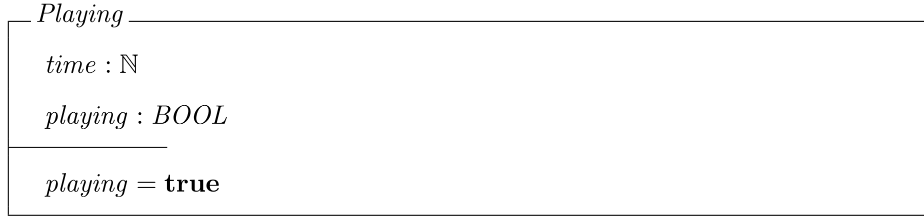
concatenate the beforestate and afterstate with the input and output observation values from the transition label. So a transition can be written as $s_{bs} \bigstar s'_{as} \bigstar ?AOp \bigstar !AOp$. s_{bs} and s_{as} are bindings in Q. $?AOp$ and $!AOp$ are bindings with schema types matching the operation shown in the transition label, restricted to input and output observations respectively. The value of these observations are also found in the transition label. For example, the transition $(0, 2, \mathit{Jump}\ 2) = \langle n \mapsto 0 \rangle \bigstar \langle n \mapsto 2 \rangle' \bigstar \langle j? \mapsto 2 \rangle$. This lets us build δ, which matches the operations with the relevant transition bindings.

$$\delta = \{(\mathit{Inc}, \{\langle n \mapsto 0, n' \mapsto 1 \rangle, \langle n \mapsto 1, n' \mapsto 2 \rangle\}),$$
$$(\mathit{Jump}, \{\langle n \mapsto 0, j? \mapsto 0, n' \mapsto 0 \rangle, \langle n \mapsto 0, j? \mapsto 1, n' \mapsto 1 \rangle, \langle n \mapsto 0, j? \mapsto 2, n' \mapsto 2 \rangle\})\}$$
$$q_0 = \langle n \mapsto 0 \rangle$$

However, this formalisation will not work for some of the example visualisations we intended to include. For example, when diagram states are merged and the diagram state represents several states from the specification state space. Figure 6.15 is a restricted visualisation of the stopwatch specification that shows the *Tick* and *Pause*/*Play* operations. $[\mathit{time} : \mathbb{N}, \mathit{playing} : \mathit{BOOL}]$ is the specification state space but the visualisation does not have unique states for each time value. We cannot represent the state as a binding in the same way we did previously, as bindings map observations to single values. Instead, we recognise that a state like *Paused* is satisfied by multiple bindings: $\langle \mathit{playing} \mapsto \mathbf{false}, \mathit{time} \mapsto 0 \rangle, \langle \mathit{playing} \mapsto \mathbf{false}, \mathit{time} \mapsto 1 \rangle, \langle \mathit{playing} \mapsto \mathbf{false}, \mathit{time} \mapsto 2 \rangle$ and so on. So, we can represent the states as a collection of bindings, rather than a single binding. Schemas are sets of bindings and so can be used to represent the states and transitions.

We can write these states as schemas rather than an infinitely long lists of bindings.

```
┌─ Paused ─────────────────────────────
│ time : ℕ
│ playing : BOOL
├──────────
│ playing = false
└──────────────────────────────────────
```



Transition and operation schemas can then be built using schema calculus. This visualisation has inconsistent labels. Two have names while two have mathematical statements. Our first example creates the *Pause* schema, named after the transition label in the visualisation. This transition begins in the *Playing* state and ends in the *Paused* state.

$$Pause == Playing \land Paused'$$
$$Play == Paused \land Playing'$$

These transitions each show part of the *Pause/Play* operation from the specification. We can combine these transition schema to create a schema that matches the *Pause/Play* operation.

$$Pause/Play == Pause \lor Play$$

Next, we create schemas for the *Tick* transitions. These transitions do not have names in the visualisation so their schemas will be given generic names. The mathematical statements in the labels are first contained in schemas and then combined with the state schemas.

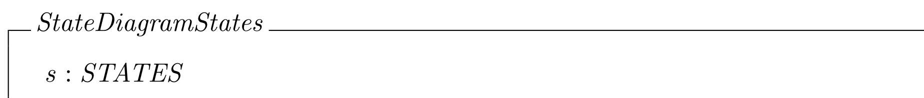
$$TickTransition1 == Playing \land Playing' \land [time, time' : \mathbb{N} \mid time' = time + 1]$$
$$TickTransition2 == Paused \land Paused' \land [time, time' : \mathbb{N} \mid time' = time]$$

These transition schemas can again be combined to create an operation schema that matches the specification.

$$TickOperation == TickTransition1 \lor TickTransition2$$

In Figure 6.15 the state schemas were named after the label of the state and contained observations from the specification state space. In this section we introduce a simpler formalisation that can be used to formalise a greater variety of state diagrams. We introduce a simple state schema with only one observation, s. s has type $STATES$, where $STATES$ is the set of states in the state diagram.

$$[STATES]$$



Now, we can then define *Paused* in Figure 6.15 as follows, the observation s has the value $Paused$.

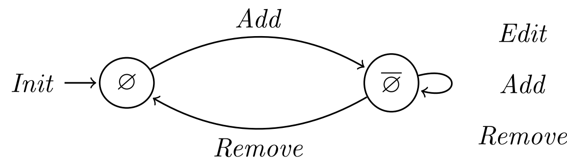


Figure 6.1: The book is empty in State1, the book is not empty in State2 (repeated from page 58)

By introducing *StateDiagramStates* and [*STATES*] we no longer need to create individual schemas for each state that include the specification observations. Despite being greatly simplified, these state schemas can still be used to build the operation schema. Each visualisation state can now be represented by a schema with the same form as *Paused*. Although the schema names are changed to match the state label these schemas are removed for brevity as their content is basically unchanged from the above example.

We use this approach to create state and operation schemas for the visualisation in Figure 6.1. We presented this state diagram earlier in the chapter, and used it to present the classic form of a state diagram. We use it here to help explain the binding and schema forms of a state diagram. First, we try to write this visualisation using bindings. For example, states can be written as $\langle book \mapsto \varnothing \rangle$ or $\langle book \mapsto \{\text{Alan Turing} \mapsto \text{19th July}\} \rangle$, while transitions are written as $\langle book \mapsto \varnothing, book' \mapsto \{\text{Alan Turing} \mapsto \text{19th July}\} \rangle$.
The initial state is written like $\langle book' \mapsto \varnothing \rangle$.
In this example, State2 represents multiple values. This is every value where the birthday book set is not empty.
$\langle book \mapsto \overline{\varnothing} \rangle$ is not a valid binding because $\overline{\varnothing}$ is not of the correct type. This means that we cannot use bindings to formalise this state diagram. We could change the formalisation such that the state is represented by multiple bindings, however, this would require us to write a large, or possibly infinite, number of bindings. So, it is easier to formalise the state diagram using the schema semantics.

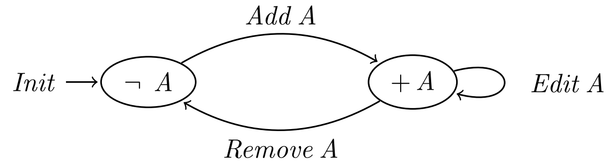


Figure 6.9: Add, Edit, Remove A (repeated from page 65)

$AddTransition1 == State1 \wedge State2'$

$AddTransition2 == State2 \wedge State2'$

$AddOperation == AddTransition1 \vee AddTransition2$

$RemoveTransition1 == State2 \wedge State2'$

$RemoveTransition2 == State2 \wedge State1'$

$RemoveOperation == RemoveTransition1 \vee RemoveTransition2$

$EditOperation == State2 \wedge State2'$

$Init == State1'$

These operation schemas can be simplified even further by inspection. For example, because the Add operation will always result in the system being in State2 (because the book is not empty when a name is added), it can be simplified to $AddOperation == State2'$.

Figure 6.9 also shows a previous visualisation, where the transitions include an input value. We use this example to show how input observations can be included in the schema form of the state diagram.

There are multiple ways to include the input observation values. First, we can add the input observation values by creating a schema that contains the information and using schema conjunction.

$$Add == State1 \wedge State2' \wedge [\, name? : NAME \mid name? = Alan \,]$$

Alternatively, we can use schema inclusion rather than schema conjunction to build the *Add* schema using the state diagram state schema:

If we include the state schema we do not need to write schemas for individual states. However, the result may be slightly less readable.

Add

$\Delta StateDiagramStates$

$name? : NAME$

$s = state1$

$s' = state2$

$name? = Alan$

The above three *Add* schemas are equivalent so the choice of which to use can be based on preference.

It is useful to build the visualisation operation schemas such that the input and output observations match the specification operations. In this example, *AddFriend* has two input observations, $name?$ and $date?$. So although this visualisation does not show $date?$, we can still include it in the schema. This makes it easier to compare the *Add* operation with the original *AddFriend* operation, as they would share the same signature.

$$Add == State1 \wedge State2' \wedge [\, name? : NAME, date? : DATE \mid name? = Alan \,]$$

6.3 Summary

In this chapter we have introduced state diagrams by showing examples of different possible types. From infinite state diagrams that would visualise the entire state space of an infinite specification if we could only draw it completely, to visualisations with merged states and unexplored states. Each of these visualisations could be used to help validate our specifications. Some focus on particular parts of the specification while others present the whole specification in new ways.

As we progressed through the chapter we formalised these state diagrams. We showed how the state diagrams could be formalised using Z bindings, Z schemas, and schema calculus. Refinement relations can then be found using the specification and the formalisation of the visualisation. This allows us to characterise the soundness of the formalisation of visualisations. When the formalisation is obvious or the meaning of the visualisation is defined by its formalisation, then we directly say that we characterise the soundness of the visualisation.

Chapter 7

Restrictions

When we want to validate some property of a system we are typically not required to visualise the entire system. Often we will only need to see a small set of states or look at one of the operations in detail. Despite this, if we explore too little of the system, then we may not observe all relevant behaviour. For example, if we are visualising a particular state, we should see all outgoing transitions. Additionally, we should not see any outgoing transitions that are not possible in the specification.

In the previous sections we looked at state diagrams that visualised the entire specification. In this section, we look at state diagrams that only visualise part of the specification. There are many reasons for not visualising the entire specification. Firstly, as we have seen when visualising the entire state space of the specification, complete visualisations can become too large to be usable. Z specifications allow for infinitely large state spaces which can make the full state space impossible to explore. Secondly, if we are generating the visualisation using some program, there are technical limitations that mean that we cannot create visualisations of the entire state space. Finally, focusing on visualising part of the system can improve clarity by removing unnecessary details. In Figure 6.9, a visualisation of the birthday book, we only visualise the operations on one person, A, leaving out the uncountably many other people that the system could include.

Another technique we investigate is using 'unexplored' or 'unexamined' states in our visualisations, such as in Figure 6.13. This is a direct representation of something found in animated visualisations and simulations. When simulating a specification the user explores the system one state at a time until the user has validated or invalidated the system. The remaining states are left unexplored. In our state diagram visualisations we include a representation of these unexplored states.

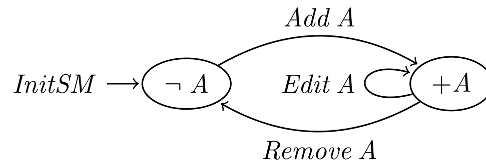


Figure 6.9: Visualising A in the birthday book (repeated from page 65)

7.1 Proving Soundness with Standard Methods

We begin by demonstrating that our standard methods are not sufficient to prove that a visualisation of part of the system is sound. If a state where the operation is enabled in the specification is not visualised then the operation will have a stronger precondition in visualisation because it is enabled in fewer states. When looking for a refinement relation we check that the visualisation operations do not have stronger preconditions than the respective specification operations. This means that, by our current definition of soundness, visualisations that do not visualise all states with enabled operations are not sound.

This is correct as anyone using such a visualisation while assuming that it visualised the entire system would be misled. For example, if Figure 6.9 was used to validate that birthday book specification, the user could assume that the birthday book can only contain one name. Any visualisation that does not visualise the entire system should not meet our current definition of soundness.

However, such visualisations are still useful. Indeed, they are often more useful than large visualisations that show the entire system. Additionally, in practice, users are unlikely to be misled if it is clear what is being visualised. So, in this chapter, we investigate how to prove if a visualisation of part of the system is sound by providing a different approach based on our standard methods.

7.2 Proving Soundness with Restrictions

Partial visualisations only visualise part of the specification. All properties and behaviour the specification has outside of what is being visualised are not considered relevant. Previously, in chapter 5, we required that for a visualisation to be sound all operations that could be used in a specification state must also be shown in the visualisation state. However, we should only check if the visualisation is an accurate representation of the part of the specification that we are choosing to visualise. For example, if we are creating a visualisation to show what operations are possible from the initial state of a specification then showing how the operations behave from other states in the specification is unnecessary and may be excluded without misleading the user.

We can create a restriction of the specification to match the partial visualisation. If we

are visualising a set of states then we restrict the specification to those states. If we are visualising how the system responds to certain inputs then we restrict the system to those inputs. Similarly, we can restrict the operations that can be used or apply any combination of these restrictions. This lets us create a restricted specification of exactly what we are verifying or demonstrating in the partial visualisation.

7.3 A New Definition of Soundness

To allow for partial visualisations we need to update our definition of soundness because the partial visualisations do not visualise all features of the specification. Previously, we used the Principle of Substitutivity to justify using refinement to characterise soundness. How do partial visualisations affect this principle? Users test the concrete system by using any abstract operations from any state and observe the changes in the expected behaviour of the system. Now instead of seeing a visualisation of the whole system, they can only see part. So, if a state (or operation, etc.) is not part of the partial visualisation then the user will not use it in order to try and discover a substitution. The users will now check if there are changes using the *restricted* abstract operations. For a partial visualisation to be sound it still needs to be accurate and not mislead the user. All transitions in the partial visualisation must still be in the specification. However, we now require that only the operations enabled in the restricted specification states must also be enabled in the corresponding state of the visualisation. For example, in the birthday book specification we can add the name B in the initial state. However, this is outside of our restriction for Figure 6.9 as we are only visualising the name A. This means that we do not require that the *Add* transition with input B is enabled in the initial state of our visualisation.

We look for a refinement relation between the restricted specification and the partial visualisation. If one is found, then we conclude that the visualisation is sound. This is valid as we are effectively changing the specification that is being visualised to match what the user expects. For Figure 6.9 this means changing the birthday book specification to only input the name A.

7.4 Restrictions

We can restrict various aspects of the specification, from the states we are visualising to the inputs of the operations. However, these restrictions can all be applied to the specification in one way: by adding to the constraints of the operations. For example, consider the following schema that sets the value of a to the value of input $i?$.

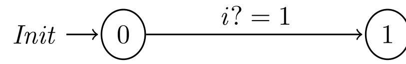
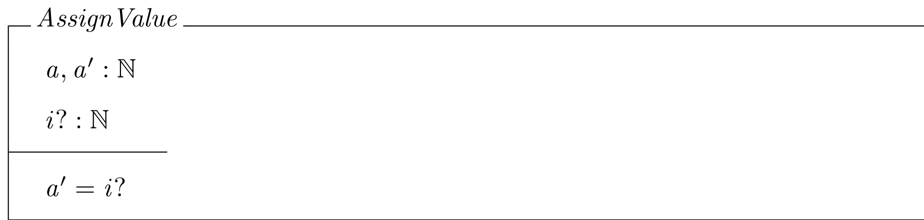


Figure 7.2: Setting a to 1



Trying to visualise the entire state space of this specification would require infinite states and transitions. Instead, we create a visualisation that only shows the operation being used with input 1 when a is in the initial state 0. This is shown in Figure 7.2 where we are only visualising two states and one input.

Next, we apply these restrictions to the specification itself. The restriction we use is $[a : \mathbb{N}, i? : \mathbb{N} \mid i? = 1 \land a = 0]$. We could also include the constraint $a' = 1$. However, we see below, in the restricted schema $AssignValue_R$, that this is unnecessary.

In the following section we look for refinement relations between the visualisation and these two schemas.

7.5 Refinement Comparison

A refinement relation does not exist between the original specification and the visualisation because the precondition of the operation has been strengthened. However, we show this by working through the proof as it is useful as an example. We cannot find refinement relations between other partial visualisations and their unrestricted specifications for the same reason. We will use simple operation refinement which means we must check for correctness and applicability.

We begin by showing correctness holds:

$$\forall State; \ State'; \ ?AOp; \ !AOp \bullet \mathbf{pre}\, AOp \land COp \Rightarrow AOp$$

$\equiv$(Substitution)

$$\forall\, a, a', i? : \mathbb{N} \bullet a = 0 \land i? = 1 \land a' = 1 \Rightarrow a' = i?$$

$\equiv$

$$\forall\, a, a', i? : \mathbb{N} \bullet a = 0 \land i? = 1 \land a' = 1 \land a' = i? \Rightarrow a' = i?$$

$\equiv (Q \land P \Rightarrow P)$

true

The visualisation satisfies the correctness requirement for *AssignValue* because the transition shown in the visualisation is also in the specification. However, the applicability property does not hold, as per the proof below, as it only visualises one transition and the precondition has been strengthened.

$$\forall\, State;\ ?Aop \bullet \mathbf{pre}\, AOp \Rightarrow \mathbf{pre}\, COp$$

$\equiv$(Substitution)

$$\forall\, a, i? : \mathbb{N} \bullet \mathbf{true} \Rightarrow a = 0 \land i? = 1$$

$\Rightarrow$ (Universal instantiation $a = 1$)

$$\forall\, i? : \mathbb{N} \bullet \mathbf{true} \Rightarrow 1 = 0 \land i? = 1$$

$\equiv$

$$\mathbf{true} \Rightarrow \mathbf{false}$$

$\equiv$

false

This statement is **false** for all instances where $a \neq 0$ and $i? \neq 1$. Therefore the refinement does not hold and by our original definition this is not a sound visualisation. However, if we instead look for a refinement relation between the visualisation and the restricted specification correctness is unchanged:

$$\forall\, State;\ State';\ ?AOp_R;\ !AOp \bullet \mathbf{pre}\, AOp_R \land COp \Rightarrow AOp_R$$

$\equiv$(Substitution)

$$\forall\, a, a', i? : \mathbb{N} \bullet a = 0 \land i? = 1 \land a' = 1 \Rightarrow a' = i?$$

$\equiv (Q \land P \Rightarrow P)$

true

However, the applicability property is satisfied because the precondition of $AssignValue_R$ is the same as the visualisation:

$$\forall State;\ ?AOp_R \bullet \mathbf{pre}\, AOp_R \Rightarrow \mathbf{pre}\, COp$$

$\equiv$(Substitution)

$$\forall a, i? : \mathbb{N} \bullet a = 0 \land i? = 1 \Rightarrow a = 0 \land i? = 1$$

$\equiv (P \Rightarrow P)$

$$\mathbf{true}$$

Both correctness and applicability properties are satisfied for the restricted specification and, therefore, the state diagram is a sound partial visualisation.

7.6 Restriction Strengths

Although we have shown that Figure 7.2 is a sound partial visualisation it may still be misleading. Firstly, the specification has been strongly restricted meaning that very little of the specification is actually being visualised. Secondly, it is not clear from the visualisation what the restriction is. Exactly how the visualisation is interpreted depends on the user and not all users will make the same assumptions concerning what is being visualised.

Like preconditions, our restrictions can be stronger or weaker compared to other restrictions. Conceptually, having a stronger restriction means we are visualising a smaller part of the specification. This allows us to choose exactly what we want to visualise, giving smaller and more focused visualisations. Having a weaker restriction means that we are visualising more of the specification, and the weakest restriction, **true**, will be visualising the entire specification.

If we use the strongest restriction, **false**, then any visualisation would be sound, as we are ignoring the specification entirely. This can be seen as the precondition of AOp_R is **false** in the applicability and correctness properties. This results in $\mathbf{false} \Rightarrow P \equiv \mathbf{true}$:

$$\forall State;\ ?AOp_R \bullet \mathbf{false} \Rightarrow \mathbf{pre}\, COp$$
$$\forall State;\ State';\ ?AOp_R;\ !AOp \bullet \mathbf{false} \land COp \Rightarrow AOp_R$$

Because of this we define the strongest reasonable restriction that could be used. If a transition is shown in the visualisation it must be in the restricted specification. For each operation in the specification we add constraints based on the transitions for that operation in the visualisation. For the state diagram visualisations that we have been investigating there is a simple method to find these constraints. We start by writing the visualisation in schema form, as we have done previously. This gives us our restriction. For each operation

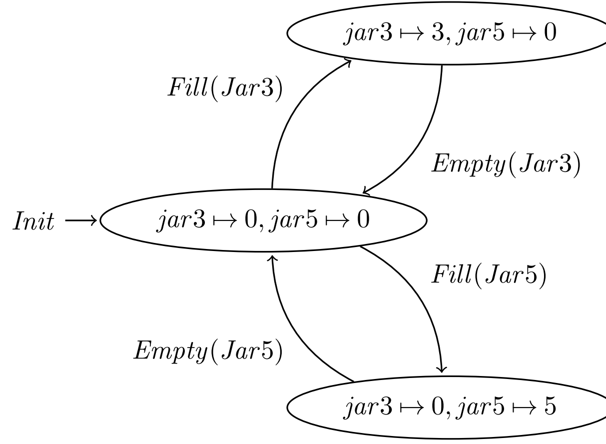


Figure 7.3: Filling and emptying the jar 1

we combine the visualisation schema with the specification schema. This method produces a set of restricted operation schemas that we can then use to prove the soundness of the restricted visualisation.

For example, we find the strongest reasonable restriction for the *Fill* operation in Figure 7.3. This will restrict the operation to the two *Fill* transitions in the visualisation. Although the state labels include mappings we need to write the restriction using the specification observations. How the states in the visualisation are related to the specification observations should be clear. If not, then the ambiguous visualisation should be clarified.

Restriction

$\Delta Level$

$j? : Jars$

$level(j3) = 0 \wedge level(j5) = 0 \wedge level'(j3) = 3 \wedge level'(j5) = 0 \vee$

$level(j3) = 0 \wedge level(j5) = 0 \wedge level'(j3) = 0 \wedge level'(j5) = 5$

Simple schema conjunction lets us build the restricted specification schema.

$Fill_Jar_R$

$\Delta Level$

$j? : Jars$

$level(j3) = 0 \wedge level(j5) = 0 \wedge level'(j3) = 3 \wedge level'(j5) = 0 \vee$

$level(j3) = 0 \wedge level(j5) = 0 \wedge level'(j3) = 0 \wedge level'(j5) = 5$

$level(j?) < max_fill(j?)$

$level' = level \oplus \{j? \mapsto max_fill(j?)\}$

If we use the strongest reasonable restriction and the refinement exists then the visualisation is sound and every transition in the visualisation must also be in the specification.

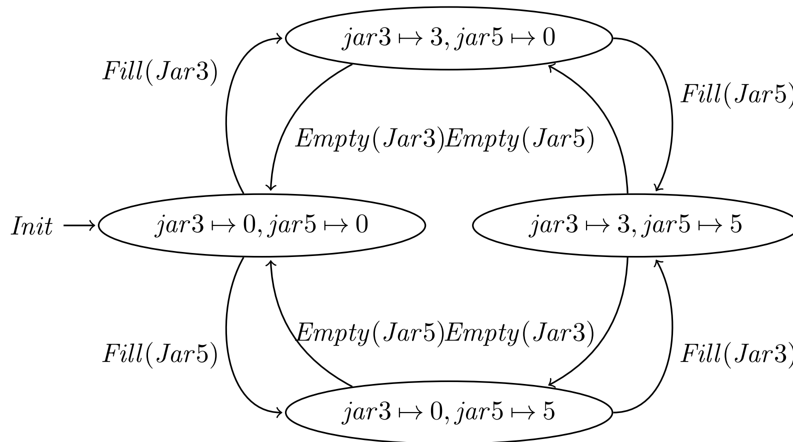


Figure 7.4: Filling and emptying the jar 2

Weakening this restriction means that we are visualising more of the specification. If the visualisation is no longer sound because we are using a weaker restriction then there must be a transition in the specification that we are not visualising properly in the visualisation. However, if the restriction is weaker and the partial visualisation is sound then we are showing that transitions that could be in the restricted specification do not actually exist. Demonstrating that operations are not enabled in the specification is useful.

In this set of examples we will be visualising the *Init*, *Fill*, and *Empty* operations of the Jars specification. The visualisation in Figure 7.3 shows the jars being filled and emptied back and forth from the initial state where both jars are empty. We can prove that the visualisation is sound using the strongest reasonable transition restriction which means that these four transitions all exist in the specification. However, if we try to use a weaker restriction such as restricting the specification to the states shown in the visualisation then the visualisation is not sound. This means that there are transitions outgoing from these states that we are not visualising. In appendix I we apply these two restrictions and show how this changes the result of the refinement properties.

The next visualisation in Figure 7.4 adds the extra state where both jars are full. Again, this visualisation is sound with the strongest reasonable transition restriction as all the transitions here exist in the specification. If we weaken this restriction to the strongest state restriction, which means that we are visualising the states with only empty or full jars, then this is still a sound visualisation. So, these are the only *Add* and *Remove* transitions outgoing from these four states. If we tried to weaken the restriction any further to include other states then this visualisation would no longer be sound simply because it does not show the transitions from other states.

The visualisation does not explicitly specify what is being visualised. If the user believes that the partial visualisation was actually a visualisation of the entire system (i.e. where the restriction is **true**), or a visualisation of a different part of the system, then their

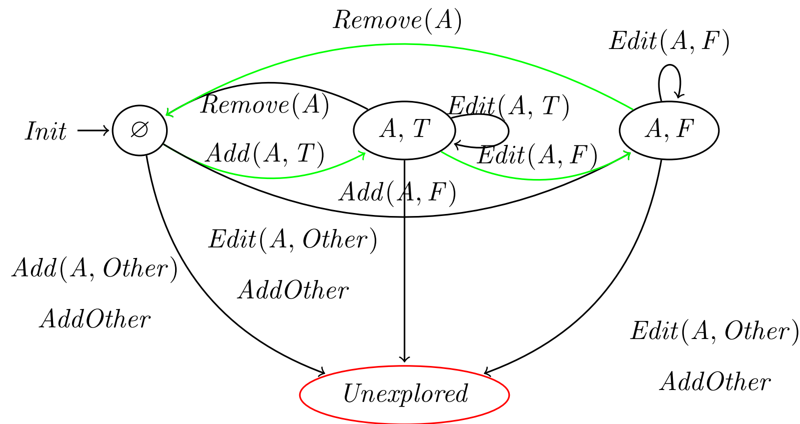


Figure 6.13: Unexplored Alan Turing Visualisation (repeated from page 68)

interpretation of the visualisation would not be sound. However, when the user uses the refinement method to determine if the visualisation is sound they choose the restriction they believe is correct and so are actually testing if their interpretation is sound.

If what the state diagram is visualising has not been described, either informally or formally, then it is useful to have a default interpretation. We use the strongest state restriction which restricts the specification to the states that have outgoing transitions in the visualisation. This means that if our visualisation is sound we know that we have completely examined every state in the visualisation. If a transition exists then it must also be possible in the specification and if it does not exist then it is not enabled in the specification. The following section focuses on state diagrams that use the state restriction as default.

7.7 Examined and Unexamined States

A simple way to build smaller visualisations is to only examine certain states in the specification. These states can be chosen in various ways. For example, we can choose states that share a similar property. Other states from the specification are left unexamined. This is similar to the approach taken by the *ProZ* state diagram representation of the simulation. This simulation starts by initialising the system and building the state diagram by exploring states one state at a time. Only states with incoming transitions are displayed and states only have outgoing transitions after they have been explored. Each state displays the values of the observations in the state. When the user has decided they have seen enough the simulation will have generated a state diagram that visualises part of the specification.

This is not the only way to visualise part of a specification. There are many details that can be changed depending on the situation in order to get a more appropriate visualisation. For example, we don't need to start in the initial state.

The example in Figure 6.13 is a visualisation of a name being added to the birthday

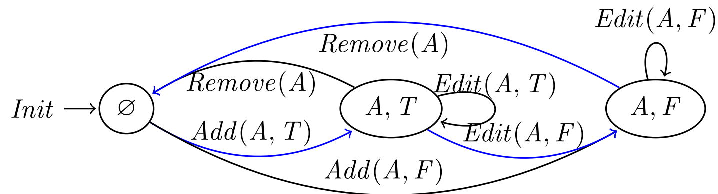


Figure 7.6: Add, Edit, Remove without Unexplored

book, then edited, then removed. This visualisation explores three states which are: when the book is empty, when it includes only name A with date T, and when it includes only name A with date F. Additionally, the state diagram includes an *Unexplored* state which includes states that were not explored.

When we examine states in the specification we visualise all outgoing transitions from those states. This shows the possible behaviours of the operations in the specification in the examined states. Not all outgoing transitions lead to other explored states however our state diagram requires end states for all transitions. We introduce the *Unexplored* or *Unexamined* states that are used as end states for these transitions. It is drawn in red to show that it is different from the examined states as we do not visualise the behaviour of the system while it is in the *Unexplored* state.

Because including the operations in full takes up a lot of space on the page we have simplified them slightly here without losing any information. The most notable simplifications are those that include the word *Other*, which means either inputting a date that is not T or F, or adding a name which is not A. We show that this is a sound partial visualisation in appendix G.

When the *Unexplored* state is removed from this visualisation it is no longer a sound visualisation of the three remaining states as we can see in Figure 7.6 and appendix J. Because we have removed the *Unexplored* state, we have also removed the transitions entering it, meaning that this is not a partial visualisation of the three states it shows.

The *Unexamined* state represents states that we are not interested in examining. It is used as an afterstate for any transitions that start inside the restriction and end outside. This allows us to construct state diagrams that can fully examine the relevant states.

Although the example in Figure 6.13 has merged all unexplored states into a single state this is not a requirement. Figure 7.7 is a visualisation of two states of the *Jars* example which has left the unexplored states unmerged to clearly show the afterstates of all transitions. The states are coloured red to indicate to the user that these states have not been explored and as such the lack of outgoing transitions is expected and not an indication of the behaviour of the system in these states. Proofs showing this visualisation is sound can be found in appendix K.

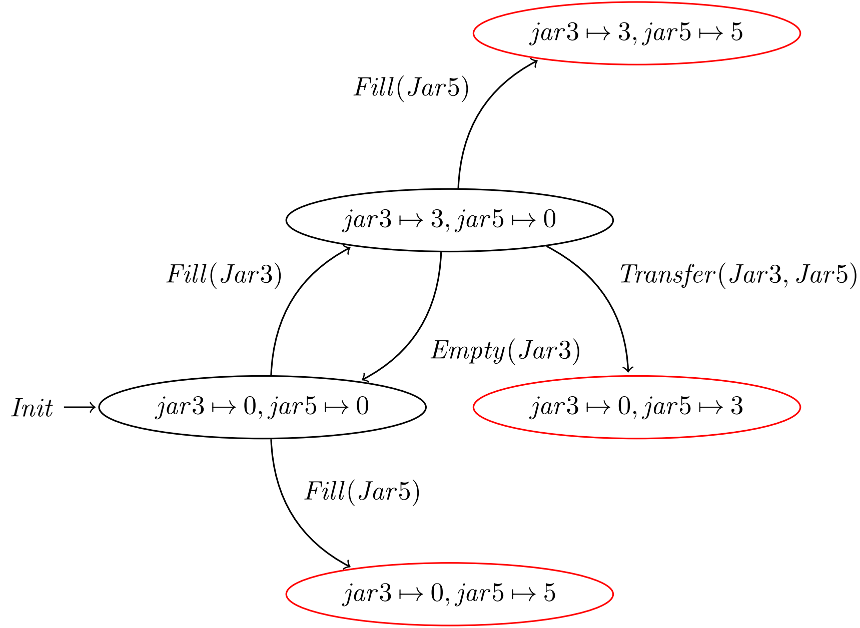


Figure 7.7: Filling and emptying the jar 3

7.8 Using Special State $\uplus$

Apart from the *Unexamined* state we have not changed how the state is represented. Rather than using data refinement we could choose to use the simpler Woodcock operation refinement rules from section 4.4. As an example of using the power of Z_c we formalise the idea of the *Unexamined* state and show that certain simple properties hold. We use this to extend the Woodcock operation rules to include the *Unexamined* state. U_R represents the restricted operation U. U_R only contains transitions outgoing from states that are inside the restriction. These transitions are the same as they were in U, unless they would terminate in states outside the restriction. In this case, the endstate of the transition is changed to $\uplus$.

In order to define the special state $\uplus$, we must first define new types, terms, and values. Any term with the type $\uplus$ has value $\uplus$.

$t^{\uplus} =_{df} \uplus$

Since we will most commonly be using $\uplus$ to refer to the unexamined state, we will define a binding with this name.

$\uplus^{[\uplus:\uplus]} =_{df} \langle \uplus \mapsto \uplus \rangle^{[\uplus:\uplus]}$

We will also need a similar binding to represent the afterstate. The definition below includes labels that will help explain these definitions.

$\uplus'^{[\uplus'_{[2]}:\uplus_{[3]}]_{[4]}}_{[1]} =_{df} \langle \uplus'_{[2]} \mapsto \uplus_{[5]} \rangle^{[\uplus':\uplus]_{[4]}}_{[6]}$

We define the term $\uplus'_{[1]}$ as a binding $\langle \uplus' \mapsto \uplus \rangle_{[6]}$ which has observation name $\uplus'_{[2]}$ with the value $\uplus_{[5]}$, which is a value unique in the type $\uplus_{[3]}$. The type of the binding (and the term being defined), is the schema type $[\uplus' : \uplus]_{[4]}$. Bindings of this type have a single observation $\uplus'_{[2]}$, where $\uplus'_{[2]}$ has type $\uplus_{[3]}$. We have named the term $\uplus'_{[1]}$ instead of using the meta-

variable t, since the schema type of $\uplus'_{[1]}$ has a unique binding, so we can give the term a unique name.

The difference between the two definitions for the terms $\uplus$ and $\uplus'$ is that $\uplus'$ is primed, which signifies that this is an afterstate. Now that $\uplus$ and $\uplus'$ have been properly defined, we can use them as bindings without explicit typing in the proofs below.

To allow our visualisation and our operation U_R to include the unexamined state it must have a schema type that can have $\uplus$ and $\uplus'$ as a possible states in addition to regular states.

$$T^{\uplus} =_{df} T^{in} \curlyvee T^{out} \mid T^{in} \curlyvee [\uplus' : \uplus] \mid [\uplus : \uplus] \curlyvee T^{out} \mid [\uplus : \uplus] \curlyvee [\uplus' : \uplus]$$

Below are the introduction and elimination rules for U_R. $\uplus+$ is used to lift transition bindings such that $\uplus$ can be a possible state. Every binding $z_0 \bigstar z_1'$ has the same before and afterstates after being lifted. Later we introduce rules that can change the afterstate to $\uplus$.

$$\frac{z_0 \bigstar z_1' \in U^{T}}{z_0 \bigstar z_1' \in U^{T^{\uplus}}} \; \uplus+$$

There are two restriction introduction rules. U_R contains all U transitions that start and end inside the restriction.

$$\frac{R.z_0 \qquad R.z_1' \qquad z_0 \bigstar z_1' \in U \qquad z_0 \bigstar z_1' \in T^{\uplus}}{z_0 \bigstar z_1' \in U_R} \; R_1+$$

The second introduction rule shows U_R contains transitions from z_0 to the unexamined state when z_1' is outside the restriction.

$$\frac{R.z_0 \qquad z_0 \bigstar z_1' \in U \qquad \neg\, R.z_1' \qquad z_0 \bigstar z_1' \in T^{\uplus}}{z_0 \bigstar \uplus' \in U_R} \; R_2+$$

We also introduce some elimination rules that may be useful. The first shows that all transitions within U_R start within the restriction.

$$\frac{z_0 \bigstar z_1' \in U_R}{R.z_0} \; R_1-$$

The second shows that if a transition $z_0 \bigstar z_1'$ from U_R doesn't end in the special unexamined state then $z_0 \bigstar z_1' \in U$.

$$\frac{z_0 \bigstar z_1' \in U_R \qquad z_1' \neq \uplus'}{z_0 \bigstar z_1' \in U} \; R_2-$$

The third shows the type of the bindings within U_R is $T^{\uplus}$

$$\dfrac{z_0 \bigstar z_1' \in U_R}{z_0 \bigstar z_1' \in T^{\uplus}}\ R_3-$$

Now we will show two cases where $U_R \subseteq U$. Firstly, if you remove $\uplus$ from U_R then the resulting operation will be a subset of U.

$$U_R \setminus \{z_0 \bigstar z_1' \in T^{\uplus} \mid z_1' = \uplus'\} \subseteq U$$

The proof of this is shown in the following tree. We introduce a binding $t_0 \bigstar t_1'$ from the restricted operation where t_1' cannot be $\uplus'$. This transition is also a transition in U, so the entire set of transitions like this must be a subset of U.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{t_0 \bigstar t_1' \in U_R \setminus \{z_0 \bigstar z_1' \in T^{\uplus} \mid z_1' = \uplus'\}}\ (1)}{t_0 \bigstar t_1' \in U_R}\ \setminus- \qquad \dfrac{\dfrac{\dfrac{}{t_1' = \uplus'}\ (2) \quad \dfrac{}{t_0 \bigstar t_1' \in U_R \setminus \{z_0 \bigstar z_1' \in T^{\uplus} \mid z_1' = \uplus'\}}\ (1)}{\mathbf{false}}}{t_1' \neq \uplus'}\ \neg + (2)}{t_0 \bigstar t_1' \in U}\ R_2-}{U_R \setminus \{z_0 \bigstar z_1' \in T^{\uplus} \mid z_1' = \uplus'\} \subseteq U}\ \subseteq +(1)$$

Secondly, if the unexamined state is not in U_R then U_R is a subset of U.

$$\dfrac{\dfrac{\dfrac{}{z_0 \bigstar z_1' \in U_R}\ (1) \qquad \dfrac{}{z_1' \neq \uplus'}}{z_0 \bigstar z_1' \in U}\ R_2-}{U_R \subseteq U}\ \subseteq +(1)$$

7.9 Lifting and Totalising

$W_\bullet$−refinement uses lifted and totalised operations to check if a refinement exists. This means that if an operation would be used outside of its precondition it will nondeterministically enter any possible state including the $\perp$ state. The lifted totalisation of a set of bindings has been defined as follows:

$$\overset{\bullet}{U} =_{df} \{z_0 \bigstar z_1' \in T^* \mid \mathit{Pre}\ U\ z_0 \Rightarrow z_0 \bigstar z_1' \in U\}$$

We now have two operations which change our sets of bindings; lifting and totalising, and restricting. We will now investigate how the order of these operations can change the resulting set of bindings.

Lifting and totalising adds a large number of transitions from every state that does not already have an outgoing transition to every state. Restricting the operation removes all transitions that start outside the restriction, and may add transitions that end in the unexamined state.

$(\overset{\bullet}{U})_R$: Lifting then restricting will add transitions outgoing from unexplored states that

Figure 7.8: $(\overset{\bullet}{U_R})$ 1

would then be removed.

$(\overset{\bullet}{U_R})$: Restricting then lifting will remove transitions outgoing from unexplored states and then add them back in.

Since we will be using $W_\bullet-$refinement we will first take the restriction then lift and totalise. This gives us the correct form to check for refinement between the restricted operation and visualisation.

7.10 $(\overset{\bullet}{U_R})$

Since we have introduction rules for both restricted operations and lifted operations we can deduce the rules needed to directly introduce $(\overset{\bullet}{U_R})$. The **pre** $-$ and $\bullet +$ rules used in these proofs can be found in Appendix A. Firstly, a binding in the original U operation that ended inside the restriction will still be in the final operation. We show an example operation U on the left of Figure 7.8. On the right we show the transitions that are introduced when the operation is restriction to states 1, 2, and 3. Note that because U is lifted we also include the states $\uplus$ and $\bot$.

$$\dfrac{\dfrac{\dfrac{}{\mathbf{pre}\ U_R\ z_0}(1) \qquad \dfrac{z_0\bigstar z_1' \in U \qquad R.z_1 \qquad \dfrac{\dfrac{}{y =_{T^{in}} z_0}(2) \qquad \dfrac{\dfrac{}{y \in U_R}(2)}{R.y}R_1-}{R.z_0} =}{z_0\bigstar z_1' \in U_R} R_1+}{z_0\bigstar z_1' \in U_R}\ \mathbf{pre}-(2)}{z_0\bigstar z_1' \in (\overset{\bullet}{U_R})} \bullet +(1)$$

Secondly, states outside the restriction have transitions to all states. This can be seen in Figure 7.9 where states $\uplus$, 4 and $\bot$ have transitions to all states while 1, 2, and 3 have

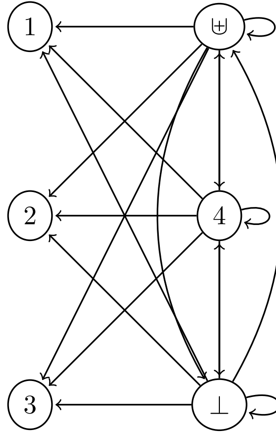


Figure 7.9: $(\overset{\bullet}{U_R})$ 2

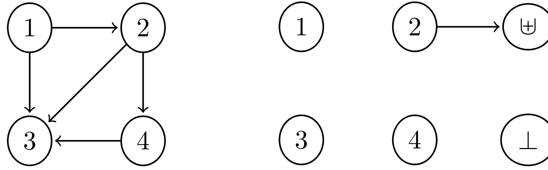


Figure 7.10: $(\overset{\bullet}{U_R})$ 3

no outgoing transitions.

$$
\frac{\dfrac{\dfrac{}{\mathbf{pre}\ U_R\ z_0}\,(1) \qquad \dfrac{\neg\, R.z_0 \qquad \dfrac{\dfrac{}{y =_{T^{in}} z_0}\,(2) \quad \dfrac{\dfrac{}{y \in U_R}\,(2)}{R.y}\,R_1-}{R.z_0}\,=}{\mathbf{false}}}{\mathbf{false}}\,\mathbf{pre}-(2)}{\dfrac{z_0\bigstar z_1' \in U_R}{z_0\bigstar z_1' \in (\overset{\bullet}{U_R})}\,\bullet+(1)}\,\mathbf{false}-
$$

Thirdly, if there is a transition in $z_0\bigstar z_1' \in U$ that ends outside the restriction then a transition from z_0 to ⊎ is included in the final operation. This can be seen in Figure 7.10. There is only one transition ending outside the restriction $\{1, 2, 3\}$ so this rule only introduces a single transition from 2 to ⊎.

$$
\frac{\dfrac{\dfrac{\dfrac{}{\mathbf{pre}\ U_R\ z_0}\,(1) \qquad \dfrac{\dfrac{}{y =_{T^{in}} z_0}\,(2) \quad \dfrac{\dfrac{}{y \in U_R}\,(2)}{R.y}\,R_1-}{R.z_0}\,=}{R.z_0}\,\mathbf{pre}-(2) \qquad \neg\, R.z_1' \qquad z_0\bigstar z_1' \in U}{z_0\bigstar ⊎' \in U_R}\,R_2+}{z_0\bigstar ⊎' \in (\overset{\bullet}{U_R})}\,\bullet+(1)
$$

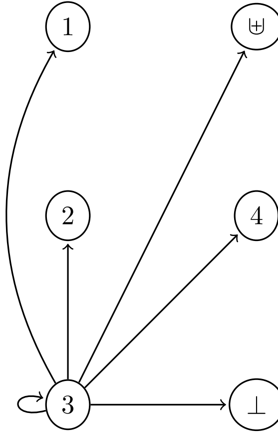


Figure 7.11: $(\overset{\bullet}{U_R})$ 4

Finally, if a state does not have any outgoing U transitions then in the final operation it will be lifted and totalised to have transitions to every state. Continuing our example, in Figure 7.11 state 3 had no outgoing transitions in U so it has transitions to all states in $(\overset{\bullet}{U_R})$.

$$\cfrac{\cfrac{\neg\ \textbf{pre}\ U\ z_0 \quad \cfrac{\cfrac{}{\textbf{pre}\ U_R\ z_0}\ (1)}{\textbf{pre}\ U\ z_0}\ \textbf{pre}\subset}{\cfrac{\textbf{false}}{z_0\bigstar z_1' \in U_R}\ \textbf{false}-}\ \textbf{false}+}{z_0\bigstar z_1' \in (\overset{\bullet}{U_R})}\ \bullet+(1)$$

7.11 Refinement

Woodcock refinement is defined as follows.

$$U_0 \sqsubseteq_{W\bullet} U_1 =_{df} (\overset{\bullet}{U_0}) \subseteq (\overset{\bullet}{U_1})$$

We will be using this to show a refinement relation exists between a visualisation and a restricted specification.

$$V \sqsubseteq_{W\bullet} U_R =_{df} \overset{\bullet}{V} \subseteq (\overset{\bullet}{U_R})$$

To show that a refinement exists we must first lift and totalise the operations in the visualisation. For each operation we check each visualisation state and see if it has an outgoing transition. If it does not, then the lifted operation would have transitions to every state in the state space as well as to ⊎ and ⊥. To show that $\overset{\bullet}{V} \subseteq (\overset{\bullet}{U_R})$ we need to show that every transition in the lifted visualisation also exists in the lifted restricted specification. However, checking these transitions individually is time consuming. Instead, for each state,

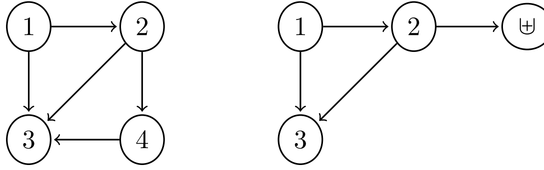


Figure 7.12: Partial Visualisation of U

we check if it is outside the precondition of the specification or outside the restriction. These states will be totalised in $(\overset{\bullet}{U_R})$ so the lifted visualisation transitions outgoing from these states will be a subset. Then, if all remaining visualisation transitions inside the restriction are in the specification, we have shown a refinement relation exists. Figure 7.12 shows a partial visualisation of our example U. Based on the work we have done in this section, to show that this partial visualisation is sound we need to check that the three visualisation transitions $1 \mapsto 2, 2 \mapsto 3$, and $1 \mapsto 3$ are in U. Additionally we need to check that a transition exists in the specification that starts in state 2 and ends outside the restriction.

This method allows us to use operation refinement for visualisations that include unexamined $\uplus$ states. We use this method in section 9.4. We have introduced this method because operation refinement is simpler than data refinement. However, most of the visualisation examples we look at have a different state space compared to the specification. Because of this we use data refinement more often than operation refinement.

7.12 Summary

By removing unimportant details partial visualisations can be used to focus on particular parts of the specification. However, the standard refinement rules cannot be used to characterise the soundness of these visualisations. We have introduced two ways to extend these rules.

Firstly, we observed that these visualisations only show a part of the specification. The specification could be restricted in many ways, such as only allowing certain inputs or being restricted to a smaller area of the state space. We changed the standard refinement rules to allow for restricted specifications and discussed how different restriction strengths can affect the soundness of the visualisation. Restrictions can be added to the constraints of the schemas in the specification. For example, the birthday book specification can be restricted to only one person by adding $name? = Alan$ to the constraints of the operation schemas.

Secondly, some visualisations contain an unexamined state that helps the user understand that only part of the specification is being visualised. We used Z_C to lift the state space of the specification and include the $\uplus$ state. Then we showed how this special state can be used while looking for a refinement relation.

Chapter 8

Microcharts

8.1 Microcharts

μ-Charts is a language that can be used to specify the behaviour of reactive systems in a graphical form. The μ-Charts language is used to create visualisations called μ-charts. Unlike the various different types of state diagram we investigated in chapter 6, μ-Charts have a precise semantics. μ-Charts were originally presented by Philipps and Scholz in [86] and are an extension of Mini-Statecharts presented in [90, 47]. These are based on the behavioural specification language Statecharts [42] which are an extension of conventional state-transition diagrams. After many developments and extensions to the language, the semantics of μ-Charts is given in Z [97, 96, 95]. μ-Charts is a "visual" specification language which we will use for visualising Z specifications.

There are two reasons we are investigating μ-Charts. Firstly, we can use the language to create more complex visualisations. Secondly, it is a type of visualisation with formally defined semantics. In Section 5.3 we discussed how formal semantics can be used as the true meaning of the visualisation. When using this philosophy we do not need to worry about misunderstandings or different interpretations of the visualisation. The main alternative language we could have chosen is UML statechart diagrams which are another variant of classical statecharts. However, its formal semantics are inadequate [28].

A μ-chart receives sets of input signals from the environment. We assume that these arrive together, perhaps each time the environment in which the chart is embedded makes a notional "tick". Receiving signals can cause the state of the μ-chart to change and when this happens it may also output signals back to the environment. All this happens (at this level of abstraction) instantaneously.

The semantics of μ-charts also lets us compose charts together, include local variables, and feed signals back into the chart that may instantaneously trigger additional transitions. These properties allow us to visualise a complex system in fewer states than state diagrams.

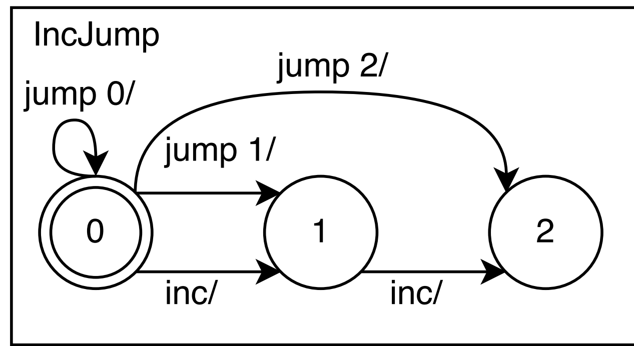


Figure 8.1: Simple Sequential μ-chart

μ-charts have been used to represent Presentation Interaction Models (PIMs), which are models of the dynamic behaviour of interface designs [19].

In this chapter we begin by providing some examples of μ-Chart visualisations. Then we provide the Z transition model for μ-Charts. We discuss some changes that will be made to the language and the purpose of these changes. Finally, we characterise the soundness of μ-Chart visualisations using refinement.

8.2 Microchart Examples

Figure 8.1 is an example of a simple sequential μ-chart. A simple sequential μ-chart does not contain any feedback and is similar to a state diagram put in a box with a name in the top left. However, the transitions are different and can be split into two parts. On the left hand side of the / in the label of the transitions there is a *guard*. This is a set of signals and when all of these signals are received from the environment, or via feedback, the transition will trigger. This will move the system into a new state and any signals listed on the right hand side of the / (the *action*) are output. The / is sometimes omitted if the label only contains a guard with no output. In Figure 8.1, if we receive the signals *jump* and 1 while we are in the initial state, state 0, then the system will transition to state 1 and output nothing. The signals *jump* and 1 are received from the environment when the *jump* operation is 'used' with input 1.

In this section we look at how μ-charts can be used to create more sophisticated visualisations than state diagrams. Let us first look how we may compose multiple charts together.

We have used composition to create Figure 8.2: a visualisation of the stopwatch specification. The top chart shows what the current time value is, while the bottom chart shows whether the stopwatch is paused or playing. The ability to compose several charts allows us to separate the different observations in the specification into separate charts. This allows us to quickly see which operations affect which observations. By removing the need to

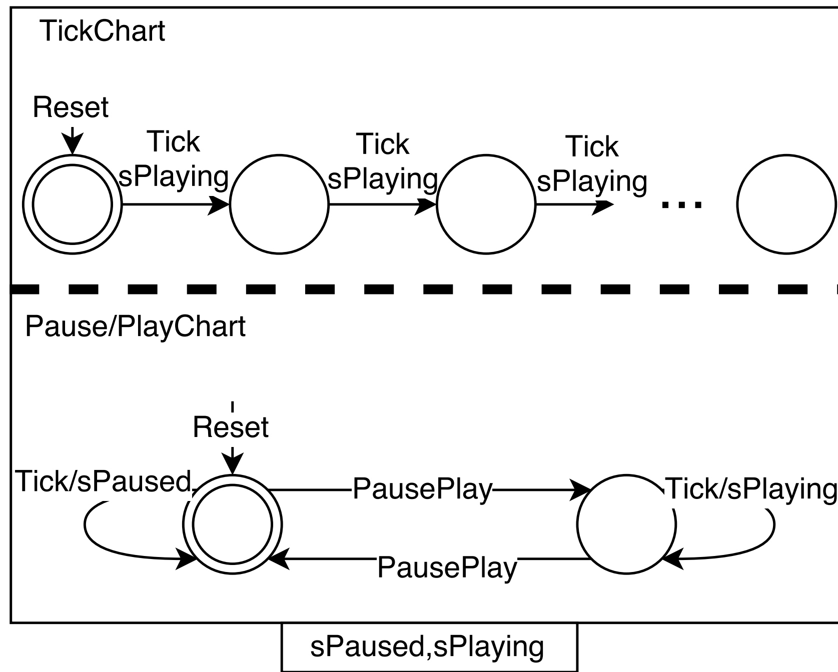


Figure 8.2: Composed Microchart

visualise every possible combination of values we can greatly reduce the number of states in the visualisation.

The bottom chart outputs feedback signal *sPlaying* which the top chart uses in its guards in some transitions to ensure that the state only changes when the stopwatch is playing. Feedback, input, and output all happen instantaneously on the same tick, so if one chart outputs a signal in a transition which is the guard of a transition in another chart then both transitions occur at the same time. The bottom chart can also output feedback signal *sPaused*. However, this signal is not used by the top chart and is only included to make the model seem more realistic and help the user understand that the stopwatch is paused. The top chart also includes some extra syntactic sugar that is not part of the formal semantics. We use ... to show that the pattern continues rather than drawing all states. The smaller box underneath the μ-chart shows any feedback signals. In this case, *sPaused* and *sPlaying*.

We can also build μ-charts that include local variables which can be seen in the top right of Figure 8.3. Instead of using a chart that visualises each possible value of time as a different state we can, as here, use a local variable to store the current value of time or any other observation in the specification. This is particularly useful for observations that have a large number of possible values such as sets or functions.

To take advantage of the local variables the guard can also include comparisons and the action can include assignments. For example, $time := 0$ resets the value of our local variable $time$ to 0. We can also use value-laden signals to assign our local variables with values from the environment. We show that this visualisation is sound in Appendix M using data refinement and the Z semantics of this μ-chart.

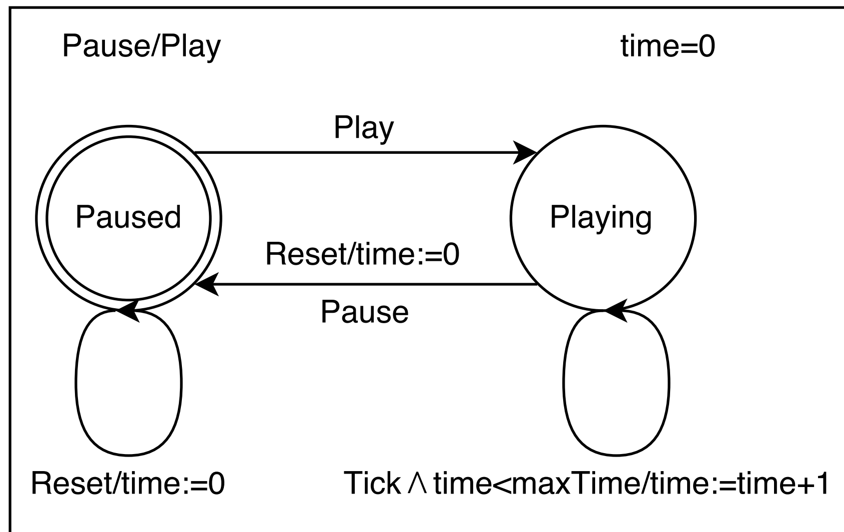


Figure 8.3: Microchart with Local Variable

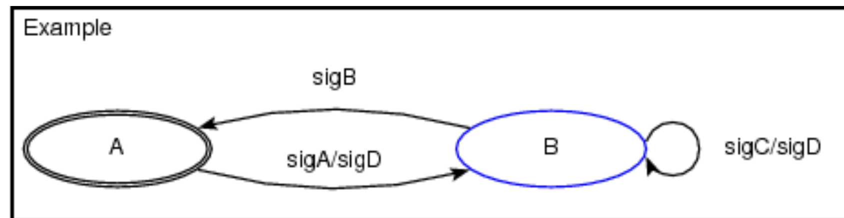


Figure 8.4: aMuZed Microchart

aMuZed is a graphical editing tool for the creation of μ-charts. Figure 8.4 shows an example μ-chart built using this tool. ZooM is an associated tool which allows users to convert their μ-charts into a Z specification [111].

8.3 The Z Semantics of MicroCharts

In this section we provide the semantics of μ-Charts. For more information see [97]. Each state and each transition in the μ-chart is written as a Z schema. Schema disjunction is then used to build a schema that describes the transition behaviour of the chart. The composition, decomposition, and hiding operators are schemas that are used to describe more complex μ-charts. Here we only provide the composition operator. Finally, the step semantics schema maps the beforestate of the entire chart and its input to its afterstate and output.

Sequential charts are described as a tuple that has the fields $(C, \Sigma, \sigma_0, \Psi, \delta)$.

- C is the name of the chart, shown in the top-left corner.
- Σ is the set of states.
- σ_0 is the initial state, drawn with a double circle.

- Ψ is the set of feedback signals.
- δ is the set of transitions in the chart.

The following axiomatic definitions encode the states, input, and output of the chart. *in C* and *out C* are functions that give the signals in all guards and actions of the chart C.

$$
\begin{array}{|l}
states_C : \mathbb{P}\, \mu_{State} \\
in_C : \mathbb{P}\, \mu_{Signal} \\
out_C : \mathbb{P}\, \mu_{Signal} \\
\Psi : \mathbb{P}\, \mu_{Signal} \\
\hline
states_C = \Sigma \\
in_C = in\ C \\
out_C = out\ C
\end{array}
$$

The Z state schema has an observation that determines the current state of the chart.

$$Chart_C \mathrel{\widehat=} [\, c_C : states_C \,]$$

The initial state of the chart is modelled by the following schema which sets the current state to the initial state of the chart.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Init_C} \\
\hline
Chart_C \\
\hline
c_C = \sigma_0 \\
\hline
\end{array}
$$

Every state in the chart has its own state schema that sets the current state c_C to the value of that state. So for all $\sigma \in \Sigma$:

$$C_\sigma \mathrel{\widehat=} [\, Chart_C \mid c_C = \sigma \,]$$

Then we give an operation schema for each transition $(S_f, S_t, guard/action)$ in the chart.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\delta_{S_f S_t}} \\
\hline
CS_f \\
CS_t' \\
i_C? : \mathbb{P}\, in_C \\
active : \mathbb{P}\, \mu_{State} \\
o_C! : \mathbb{P}\, out_C \\
\hline
active(C) \\
\rho(guard) \\
o_C! = action \\
\hline
\end{array}
$$

Here, $\rho(guard)$ is a function that produces the Z predicate that models the guard. Typically, each signal, sig, in the guard will be represented by the expression $sig \in i_C? \cup (o_C! \cap \Psi)$ which is true if chart C is receiving sig as input or feedback.

The semantics also includes a schema that models the behaviour when the chart is inactive. Only decomposed charts can become inactive so for our purposes we assume that $active(C)$ will always be **true**.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Inactive_C} \\
\hline
\Xi Chart_C \\
i_C? : \mathbb{P}\, in_C \\
active_{_} : \mathbb{P}\, \mu_{State} \\
o_C! : \mathbb{P}\, out_C \\
\hline
\neg active(C) \\
o_C! = \{\} \\
\hline
\end{array}
$$

The transitional semantics of a sequential chart can now be given by the following definition.

$$\delta_C \mathrel{\widehat=} (\bigvee \{[\![t]\!]_{Z_t} \mid t \in \delta\}) \vee Inactive_C$$

This gives a schema that is a disjunction of the transition schemas in the chart, including $Inactive_C$.

There are five observations in the transition operation schemas.

- c_C is the state of the chart before the transition occurs.
- $i_C?$ is the set of input signals offered by the environment this tick that the chart can accept.
- $active_{_}$ is the set of currently active charts. Note that this is used as a predicate, for example $active\,(C) = \mathbf{true}$ if C is in the set of active charts.
- c'_C is the state of the chart after the transition occurs.
- $o_C!$ is the output generated by the chart this tick.

As an example, the following Z schemas result from the transition model of the simple sequential chart in Figure 8.1.

$$Chart_{IncJump} \mathrel{\widehat=} [\, c_{IncJump} : states_{IncJump} \,]$$

$$\begin{array}{|l}
\hline
Init_{IncJump} \\
\quad Chart_{IncJump} \\
\hline
\quad c_{IncJump} = 0 \\
\hline
\end{array}$$

$$IncJump_0 \mathrel{\widehat=} [\, Chart_{IncJump} \mid c_{IncJump} = 0 \,]$$
$$IncJump_1 \mathrel{\widehat=} [\, Chart_{IncJump} \mid c_{IncJump} = 1 \,]$$
$$IncJump_2 \mathrel{\widehat=} [\, Chart_{IncJump} \mid c_{IncJump} = 2 \,]$$

Rather than show every transition schema we show one for each of the two operations *Inc* and *Jump*.

$$\begin{array}{|l}
\hline
\delta_{02} \\
\quad IncJump_0 \\
\quad IncJump_2' \\
\quad i_{IncJump}? : \mathbb{P}\, in_{IncJump} \\
\quad active_{:} \mathbb{P}\, \mu_{State} \\
\quad o_{IncJump}! : \mathbb{P}\, out_{IncJump} \\
\hline
\quad active(IncJump) \\
\quad jump \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \\
\quad 2 \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \\
\quad o_{IncJump}! = \{\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\delta_{12} \\
\quad IncJump_1 \\
\quad IncJump_2' \\
\quad i_{IncJump}? : \mathbb{P}\, in_{IncJump} \\
\quad active_{:} \mathbb{P}\, \mu_{State} \\
\quad o_{IncJump}! : \mathbb{P}\, out_{IncJump} \\
\hline
\quad active(IncJump) \\
\quad inc \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \\
\quad o_{IncJump}! = \{\} \\
\hline
\end{array}$$

Now we can build the operation schema that combines each of the transitions in the chart.

$$\delta_{IncJump} \mathrel{\widehat=} \delta_{00} \vee \delta_{01} \vee \delta_{02} \vee \delta_{12} \vee Inactive_{IncJump}$$

$\delta_{IncJump}$

$\Delta Chart_{IncJump}$
$i_{IncJump}? : \mathbb{P}\, in_{IncJump}$
$active: \mathbb{P}\, \mu_{State}$
$o_{IncJump}! : \mathbb{P}\, out_{IncJump}$

$(active(IncJump) \wedge$
$o_{IncJump}! = \{\} \wedge$
$((c_{IncJump} = 0 \wedge$
$jump \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \wedge$
$0 \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \wedge$
$c'_{IncJump} = 0)$
$\vee$
$(c_{IncJump} = 0 \wedge$
$(jump \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \wedge$
$1 \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \vee$
$inc \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi)) \wedge$
$c'_{IncJump} = 1)$
$\vee$
$(c_{IncJump} = 0 \wedge$
$jump \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \wedge$
$2 \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \wedge$
$c'_{IncJump} = 2)$
$\vee$
$(c_{IncJump} = 1 \wedge$
$inc \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \wedge$
$c'_{IncJump} = 2))$
$\vee$
$(\neg active(IncJump) \wedge$
$c_{IncJump} = c'_{IncJump}$
$o_{IncJump}! = \{\}))$

8.3.1 Composition Operator

μ-charts can be composed together allowing the charts to communicate using feedback signals. C_1 and C_2 can be composed into the new chart $C_1 \mid \Psi \mid C_2$. The feedback signals are in the set Ψ which is shown listed in a small box under the charts. Feedback signals can be

output from one chart which can instantaneously trigger the guards of the second chart in the same tick. For example, if information about the current state of the chart is output as a feedback signal then the behaviour of a composed chart can change.

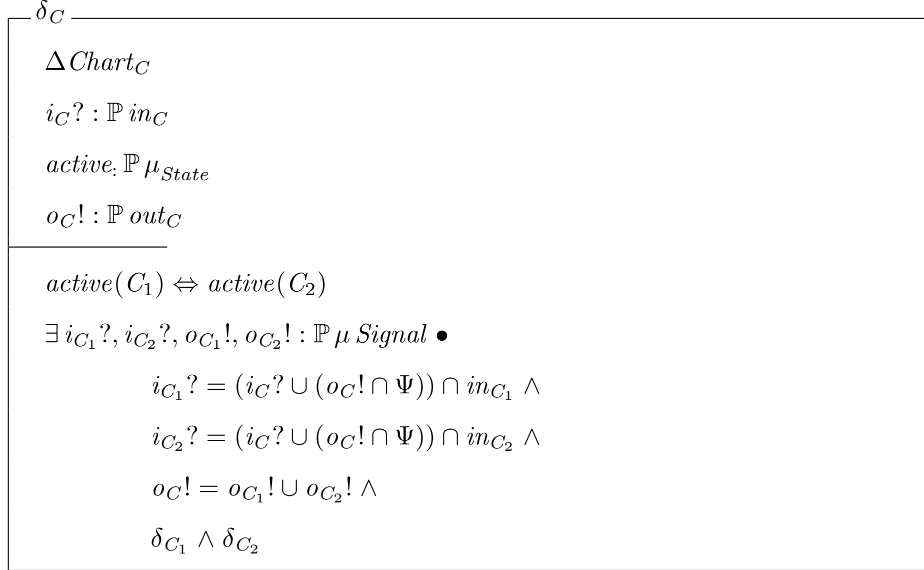
The transition model for the composed chart C is constructed recursively by first constructing the charts C_1 and C_2, which could themselves also be composed charts. The schemas of C_1 and C_2 are then combined using the following axiomatic definitions and schema to create the transition model for C.

$$
\begin{array}{l}
states_C : \mathbb{P}\, \mu_{State} \\
in_C : \mathbb{P}\, \mu_{Signal} \\
out_C : \mathbb{P}\, \mu_{Signal} \\
\Psi : \mathbb{P}\, \mu_{Signal} \\
\hline
states_C = states_{C_1} \cup states_{C_2} \\
in_C = in_{C_1} \cup in_{C_2} \\
out_C = out_{C_1} \cup out_{C_2}
\end{array}
$$

The composed chart will have a current state for each of the charts being composed. If $Chart_{C_1}$ and $Chart_{C_2}$ are both sequential charts then our composed chart state schema will have two observations; c_{C_1} and c_{C_2}.

$$
\begin{array}{l}
Chart_C \\
\hline
Chart_{C_1} \\
Chart_{C_2}
\end{array}
$$

$$
\begin{array}{l}
Init_C \\
\hline
Init_{C_1} \\
Init_{C_2}
\end{array}
$$



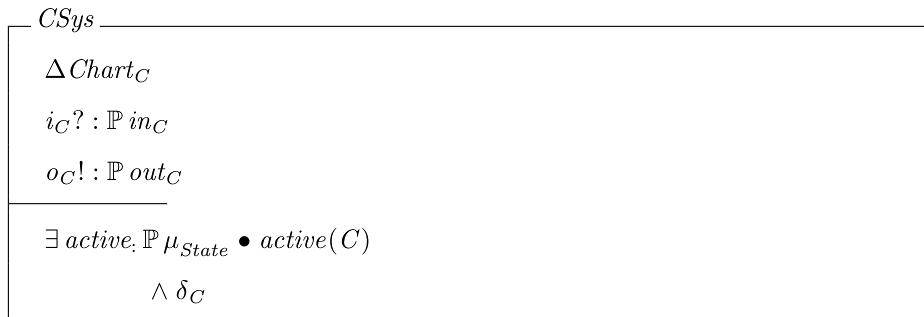
The transition model for the composed chart includes the transition models of its constituent parts. However, the signals that these parts receive is different compared to the sequential model. Instead of receiving input signals from the environment and its own feedback signals the input observations $i_{C_1}?$ and i_{C_2} now receive input values from the environment and the feedback signals that are output from either chart.

8.3.2 Step Semantics

Finally, we present the top-level schema that maps the beforestate(s) of the chart and the input from the environment to its afterstate(s) and output.

$$\mu\, Signal = in\ C$$

This schema, called *CSys*, hides the *active_* observation by ensuring the topmost chart is always active.



8.4 Soundness of Visualisations

Our goal is to prove that these visualisations, and other μ-chart visualisations, are *sound.*

If the visualisation shows that some action is possible then it must also be possible in the specification. For example, the *IncJump* visualisation in Figure 8.1 shows that we can use either the *Inc* operation or the *Jump* operation to move from state 0 to state 1. So, if this is a sound visualisation then this should also be possible in the specification and the visualisation would be unsound if either operation did not allow this in the specification. Clearly, a μ-chart that visualises incorrect or impossible behaviour is misleading and unsound.

For state diagrams we start by showing the applicability property holds. However this property always holds trivially for μ-Chart visualisations. This property would require that if an operation is enabled in the specification it must also be in the visualisation. However, μ-charts can receive a signal to use any operation in any state. Because of how this is handled in the semantics this means that **pre** $COp =$ **true**. Furthermore, when substituted into the applicability property:

$$\forall State;\ ?AOp \bullet \mathbf{pre}\ AOp \Rightarrow \mathbf{true}$$

$$\equiv$$

$$\mathbf{true}$$

However, because operations can be used in any state in μ-Chart visualisations we must be careful that they are being still used correctly by checking the correctness property holds. If all traces and changes of state in the μ-Chart visualisation are also possible in the specification then the visualisation is sound.

8.5 Changes to Microcharts

The μ-chart semantics we are using in this thesis are an adaptation of classic μ-charts as we are specifically using them for visualisations of Z specifications. The most notable difference is that we allow some syntactic sugar such as the use of ellipses for a repeating pattern and transitions with no visible start state that can be used from any state.

Additionally, since we are visualising a specification we assume that when an operation is used the operation name as well as its input observation values are sent as signals to the μ-chart. For example, if we use the jump operation to jump to state 2 we are inputting signals *jump* and 2. This allows us to use the operation name as a guard on transitions and can help make visualisations where it is clear how each operation affects the system differently. For example, the *IncJump* μ-chart could axiomatically define the signals as the following where *OperationNames* is a set containing the operation names *Inc* and *Jump* from the specification:

$$\mu\, Signals ::= Op\langle\!\langle OperationNames\rangle\!\rangle \mid N\langle\!\langle\{1,2,3\}\rangle\!\rangle$$

We can think of *Op* and *N* as functions which inject the types *OperationNames* and $\{1,2,3\}$ into $\mu\, Signals$. Additionally, any feedback signals are still included in this definition. As

such, the composed stopwatch example would define $\mu\, Signals$ as:

$$\mu\, Signals ::= Op\langle\langle OperationNames\rangle\rangle \mid sPaused \mid sPlaying$$

Similarly, instead of using a basic type we can define the type of states more explicitly. The most straightforward way of doing this is to use the set of names of states Σ. However, this can lose valuable information. For example, in the composed stopwatch example each of the states in the top chart represents a particular time value. Therefore, we can define the type as follows, where each state is either a natural number or a boolean value:

$$\mu_{States} ::= t\langle\langle \mathbb{N}\rangle\rangle \mid p\langle\langle \mathbb{B}\rangle\rangle$$

Below, we update the transition schema δ_{02} to use the changed μ Signals and μ_{States} set.

$$\delta_{02}$$

$$\begin{array}{l}
IncJump_0 \\
IncJump'_2 \\
i_{IncJump}? : \mathbb{P}\, in_{IncJump} \\
active : \mathbb{P}\, \mu_{States} \\
o_{IncJump}! : \mathbb{P}\, out_{IncJump} \\
\hline
active(IncJump) \\
Op\, jump \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \\
t\, 2 \in i_{IncJump}? \cup (o_{IncJump}! \cap \Psi) \\
o_{IncJump}! = \{\}
\end{array}$$

This is a small change syntactically, however it removes any ambiguity from the types of our values, particularly $0, 1$, and 2, which could be states, signals *or* numbers.

Finally, we need to specify how the μ-chart should behave if signals are received that do not satisfy the guards of any outgoing transitions. We can use either the *do-nothing* or the *chaotic* interpretation. Respectively, this means that either the state does not change or the state can change nondeterministically to any state. The choice here largely depends on preference and the specification being visualised as either interpretation can be used. We will be using the do-nothing interpretation as it allows us to remove some "selfloops" from the μ-chart.

The following schema describes the behaviour of a chart when the guards of all outgoing transitions are not satisfied. For example, when no input signals are received while Figure 8.1 is in state 0 no outgoing transition can be used.

When using the do-nothing interpretation the state does not change and no signals are output. This schema is used when no μ-chart transition can be used i.e for each transition $(S_{f1}, S_{t1}, guard_1/action_1), (S_{f2}, S_{t2}, guard_2/action_2)$ etc., we are not in state S_{fi} when the input satisfies $guard_i$.

$$
\begin{array}{|l}
\hline
\epsilon_C \\
\Delta Chart_C \\
i_C?, o_C! : \mathbb{P}\,\mu_{Signal} \\
active_ : \mathbb{P}\,\mu_{State} \\
\hline
active(C) \\
c'_C = c_C \\
o_C! = \{\} \\
\neg\,(S_{f1} \wedge \rho(guard_1)) \\
\neg\,(S_{f2} \wedge \rho(guard_2)) \\
\qquad\vdots \\
\hline
\end{array}
$$

This schema is then disjointed with the other transition schemas to create the total δ_C which describes how the μ-chart changes for any given state and input.

$$\delta_C == (\bigvee \{[\![t]\!]_{Z_t} \mid t \in \delta\}) \vee Inactive_C \vee \epsilon_C$$

The ϵ_C schema has been built by reverse engineering the ZooM tool which converts μ-charts into Z using the do-nothing interpretation.

8.6 Microchart Semantics

Schemas are constructed for each state, transition, and chart and the schema calculus is used to integrate these schemas into a layered system.

For example, amongst the schemas used to define a simple composed μ-chart the lower-level operation schemas specify the behaviour of the transitions in each chart being composed together. The higher-level operation schemas describe the behaviour of the system after the lower-level charts have been composed. The top-level chart, which we call *CSys*, has input and output observations for the signals to and from the environment and specifies the overall behaviour of the chart based on what signals are being received and what the current state is.

We present the *CSys* schema for the *Stopwatch* μ-chart. The low-level schemas and axiomatic definitions can be found in appendix L. The *CSys* schema predicates are divided into multiple cases based on the current state and the input signals being received from the environment. We are using *do-nothing* semantics so when we try to use an operation where there is no appropriate outgoing transition we stay in the same state. We have two state observations, c_{PP} and c_{Tick}, which observe the current state for each of the two sub-charts.

As can be seen from the *SWSys* example it is not possible to match a transition in the visualisation with a single case in the predicate part of the schema. All feedback signals that

were present in the lower-level schema have been removed and how a particular operation behaves both depends on and changes the state of each composed μ-chart. So, the first case in the predicate describes the change in states when the *Tick* operation is used while the bottom chart is in the *Playing* state and the top chart is in any state but $sTime$. When the operation is used with this precondition the bottom state stays in *Playing* and the top chart increments to the next state.

$$
\begin{array}{l}
\textit{SWSys} \\
\hline
\Delta\, \mathit{Chart}_{SW} \\
i_{SW}? : \mathbb{P}\, \mathit{in}_{SW} \\
o_{SW}! : \mathbb{P}\, \mathit{out}_{SW} \\
\hline
(\mathit{Op}\ \mathit{Tick} \in i_{SW}? \land \\
(t^{\sim}\, c_{Tick}) < \mathit{maxTime} \land \\
c'_{Tick} = t\,(t^{\sim}\, c_{Tick} + 1) \land c_{PP} = p\,\mathbf{true} \land c'_{PP} = p\,\mathbf{true} \lor \\
\mathit{Op}\ \mathit{Reset} \in i_{SW}? \land \\
c'_{PP} = p\,\mathbf{false} \land c'_{Tick} = t\,0 \lor \\
\mathit{Op}\ \mathit{PausePlay} \in i_{SW}? \land \\
c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{true} \land c'_{PP} = p\,\mathbf{false} \lor \\
\mathit{Op}\ \mathit{PausePlay} \in i_{SW}? \land \\
c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{false} \land c'_{PP} = p\,\mathbf{true} \lor \\
\mathit{Op}\ \mathit{Tick} \in i_{SW}? \land \\
c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{false} \land c'_{PP} = p\,\mathbf{false} \lor \\
\mathit{Op}\ \mathit{Tick} \in i_{SW}? \land \\
c_{Tick} = t\,\mathit{maxTime} \land \\
c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{true} \land c'_{PP} = p\,\mathbf{true}) \land \\
o_{SW}! = \{\}
\end{array}
$$

However, if we want to use Z data refinement we need to have operations in the visualisation that match the operations in the specification. For this purpose we introduce a new layer to the μ-chart semantics which may be used to separate the overall behaviour of the chart into multiple schemas that describe the behaviour of the chart when a particular operation is used.

8.7 Operation Operator

We are using μ-charts to visualise Z specifications. Hence, rather than allow the environment to input an arbitrary set of signals to the system we instead use operation names. From the

specification we are visualising we have a set of operations each with their own input and output observations. For each operation we restrict the schema to show only the changes to the system when that operation is used. We do this by ensuring that the set of input signals includes exactly the operation name as well as signals with the same types as any input observations of the operation. Similarly, we need to ensure that the output signal set matches the output observations of the operation. When using the operation no other signals should be input by the environment and the system should not output any other signals. Once we have the input and output observations we remove the signals from the schema using hiding. This ensures that the resulting schema will match the appropriate operation and we can use data refinement. If the signal observations were still present in the schema then it would not be a match for the operation defined in the specification.

The purpose of this operator is to hide the signals, and replace them with the appropriate observations for the operation. First, we look at a generic operation written in Z. This operation changes the state of the specification and has some finite set of input and output observations. Then, there is some predicate **P** that specifies how the state is changed:

The following schema illustrates how we can extract a schema with the same input and output observations as the above schema from *CSys*. μ *OperationName* will be a schema that describes a change in the state of the μ-chart rather than the specification when *OperationName* is used. In the predicate we have *CSys* where the input and output signals have been hidden. When using *OperationName* we require that the input signals include the name of the operation being used as well as a signal for each of the input observations. Similarly, we require the output signals to include exactly the output observations.

$$
\begin{array}{|l}
\hline
\mu\, OperationName \\
\Delta Chart \\
input_i? : Type_i \\
\vdots \\
input_j? : Type_j \\
output_o! : Type_o \\
\vdots \\
output_p! : Type_p \\
\hline
\exists\, i_C?, o_C! : \mathbb{P}\, \mu\, Signals \mid \\
i_C? = \{\mu\, Op\ OperationName, \mu\, i_i\ input_i?, \ldots \mu\, i_j\ input_j?\} \\
\wedge\ o_C! = \{\mu\, o_o\ output_o!, \ldots \mu\, o_p\ output_p!\} \bullet CSys \\
\hline
\end{array}
$$

In our previous examples we showed how to build the type $\mu\, Signals$. The set of signals includes the operation names, signals with the same types as the input and output observations from each of the operations, and finally feedback signals.

$$
\begin{array}{|l}
\mu\, Signals ::= \mu\, Op\langle\!\langle OperationNames\rangle\!\rangle \mid \mu\, i_i \langle\!\langle Type_i\rangle\!\rangle \mid \ldots \mid \mu\, o_j \langle\!\langle Type_j\rangle\!\rangle \mid \\
\qquad\qquad Feedback_i \mid \ldots \mid Feedback_j
\end{array}
$$

Let us apply this definition to the composed stopwatch example. The stopwatch example has no input or output observations in any of the operations:

$$
\begin{array}{|l}
\mu\, Signals ::= \mu\, Op\langle\!\langle OperationNames\rangle\!\rangle \mid sPaused \mid sPlaying
\end{array}
$$

So, when we use the *Reset* operation we expect *Reset* to be the only signal being input to the μ-chart:

$$
\begin{array}{|l}
\hline
\mu\, Reset \\
\Delta Chart_{SW} \\
\hline
\exists\, i_{SW}? : \mathbb{P}\, in_{SW}, o_{SW}! : \mathbb{P}\, out_{SW} \bullet \\
\qquad i_{SW}? = \{\mu\, Op\ Reset\} \wedge o_{SW}! = \{\} \wedge \\
\qquad SWSys \\
\hline
\end{array}
$$

Alternatively, if the original specification had input and output observations we would be inputting the appropriate signals to the μ-chart with the operation name and expect to see output signals of the appropriate type.

Below we have applied the operation operator to the *Jump* operation of the visualisation in Figure 8.1. The *Jump* schema in the specification has a single input, the number that the

state will jump to. Therefore, when we use the operation operator to determine how the *Jump* operation affects the μ-chart we know that we need to input a number from 0 to 2 called $n?$. We know that the input signals being input from the environment must be *Jump* and $n?$ if we are using the *Jump* operation. Finally, we know that there will be no output signals:

$$
\begin{array}{|l}
\hline
\mu\, Jump \\
\Delta Chart_{IJ} \\
n? : \{0, 1, 2\} \\
\hline
\exists\, i_{IJ}? : \mathbb{P}\, in_{IJ},\, o_{IJ}! : \mathbb{P}\, out_{IJ} \bullet \\
\quad i_{IJ}? = \{Op\ Jump, N\ n?\} \land o_{IJ}! = \{\} \land \\
\quad IJSys \\
\hline
\end{array}
$$

We can substitute *IJSys* and simplify this schema further to get a predicate with only three cases. If we are in state 0 we jump to the state number being input, otherwise we do not change states:

$$
\begin{array}{|l}
\hline
\mu\, Jump \\
\Delta Chart_{IJ} \\
n? : \{0, 1, 2\} \\
\hline
c = S\,0 \land c' = S\,n? \lor \\
c = S\,1 \land c' = S\,1 \lor \\
c = S\,2 \land c' = S\,2 \\
\hline
\end{array}
$$

Similarly, we can build the schema $\mu\, Inc$ by applying the operation operator to the μ-chart and using the other operation in the specification:

$$
\begin{array}{|l}
\hline
\mu\, Inc \\
\Delta Chart_{IJ} \\
\hline
c = S\,0 \land c' = S\,1 \lor \\
c = S\,1 \land c' = S\,2 \lor \\
c = S\,2 \land c' = S\,2 \\
\hline
\end{array}
$$

Using $\mu\, Inc$ and $\mu\, Jump$ along with the initialisation and state schemas from the original μ-chart semantics we can find a refinement relation between the specification and the visualisation.

The following schemas are the result of applying the operation operator to the composed stopwatch visualisation for each of the operations *Reset*, *Pause/Play*, and *Tick*. First,

$\mu\,Reset$ shows that when we use the reset operation the μ-chart will return to the paused state and the initial time 0 state:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mu\,Reset} \\ \hline
\Delta Chart_{SW} \\ \hline
c'_{PP} = cPaused \\
c'_{Tick} = t\,0 \\ \hline
\end{array}$$

$\mu\,PP$ does not change the value of time but toggles between the paused and playing states:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mu\,PP} \\ \hline
\Delta Chart_{SW} \\ \hline
c'_{Tick} = c_{Tick} \land c_{PP} = cPlaying \land c'_{PP} = cPaused \lor \\
c'_{Tick} = c_{Tick} \land c_{PP} = cPaused \land c'_{PP} = cPlaying \\ \hline
\end{array}$$

Finally, $\mu\ Tick$ increases the current time unless the stopwatch has paused or has reached the maximum time allowed. In this schema we use the relational inverse of the state observation. This lets us check that we have not yet reached the maximum time and allows us to transition into the next consecutive state by adding 1 to the current time:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mu\ Tick} \\ \hline
\Delta Chart_{SW} \\ \hline
(t^{\sim}\ c_{Tick}) < maxTime \land \\
c'_{Tick} = t\,(t^{\sim}\ c_{Tick} + 1) \land c_{PP} = cPlaying \land c'_{PP} = cPlaying \lor \\
c'_{Tick} = c_{Tick} \land c_{PP} = cPaused \land c'_{PP} = cPaused \lor \\
c_{Tick} = t\ maxTime \land \\
c'_{Tick} = c_{Tick} \land c_{PP} = cPlaying \land c'_{PP} = cPlaying \\ \hline
\end{array}$$

8.8 Refinement

In order to use Z data refinement we need operations with matching input and output observations. We now have the operation operator which gives us the visualisation formally written in Z with the input and output observations we require.

We will work through the proof of only one of the operations, *Reset*.

The retrieve relation shows how the observations in the specification are related to the states in the visualisation:

$$
\begin{array}{l}
\textit{R} \\
\hline
\textit{Stopwatch} \\
\textit{Chart}_{SW} \\
\hline
t^{\sim} c_{Tick} = \textit{time} \land \\
(c_{PP} = \textit{cPaused} \land \textit{playing} = \textbf{false} \\
\lor c_{PP} = \textit{cPlaying} \land \textit{playing} = \textbf{true})
\end{array}
$$

We also need the preconditions of our *Reset* and μ *Reset* operations. Because the *Reset* operation can be used from any state the precondition will be **true**. In fact, when we use the do-nothing μ-chart semantics the precondition of the visualisation operations will always be **true**. This is because the behaviour of the μ-chart is totally defined:

$$\textbf{pre}\, \textit{Reset} \equiv \textbf{true}$$

$$\textbf{pre}\, \mu\, \textit{Reset} \equiv \textbf{true}$$

Firstly, we check *Init*. μ *Init* can easily be found using the μ-chart semantics. The initial states in the visualisation are identified by the double circles where the time is 0 and the watch is paused:

$$\forall\, \textit{Chart}'_{SW} \bullet \mu\, \textit{Init} \Rightarrow \exists\, \textit{Stopwatch}' \bullet \textit{Init} \land R'$$

$\equiv$\{definitions\}

$$
\begin{array}{l}
\forall\, \textit{Chart}'_{SW} \bullet t^{\sim} c'_{Tick} = 0 \land c'_{PP} = \textit{cPaused} \Rightarrow \\
\exists\, \textit{playing}' : \textit{Bool}, \textit{time}' : \mathbb{N} \bullet \\
\textit{playing}' = \textbf{false} \land \textit{time}' = 0 \land R'
\end{array}
$$

$\equiv$\{one point rule, simplify\}

$$
\begin{array}{l}
\forall\, \textit{Chart}'_{SW} \bullet t^{\sim} c'_{Tick} = 0 \land c'_{PP} = \textit{cPaused} \Rightarrow \\
t^{\sim} c'_{Tick} = 0 \land c'_{PP} = \textit{cPaused}
\end{array}
$$

$\equiv$\{logic\}

true

Since the initialisation property holds we can check applicability. However, since the precondition for each visualisation operation is **true** we can quickly see that the applicability property is **true** for each operation. Here we show that for *Reset*:

$$\forall\, \textit{Chart}_{SW};\ \textit{Stopwatch};\ \bullet \textbf{pre}\, \textit{Reset} \land R \Rightarrow \textbf{pre}\, \mu\, \textit{Reset}$$

$\equiv$ {precondition of *Reset* as above}

$$\forall\, Chart_{SW};\; Stopwatch;\; \bullet\; \mathbf{pre}\, Reset \wedge R \Rightarrow \mathbf{true}$$

$\equiv$ {logic}

$$\mathbf{true}$$

The full proof checks each of the operations in the specification. We will finish with the contractual correctness proof of *Reset* as an example:

$$\forall\, Stopwatch;\; Chart_{SW};\; Chart'_{SW};\; \bullet$$
$$\mathbf{pre}\, Reset \wedge R \wedge \mu\, Reset \Rightarrow \exists\, Stopwatch' \bullet R' \wedge Reset$$

$\equiv$ {definitions}

$$\forall\, Stopwatch;\; Chart_{SW};\; Chart'_{SW};\; \bullet$$
$$\mathbf{true} \wedge R \wedge c'_{PP} = cPaused \wedge c'_{Tick} = t\,0 \Rightarrow$$
$$\exists\, Stopwatch' \bullet R' \wedge time' = 0 \wedge playing' = \mathbf{false}$$

$\equiv$ {one point rule, simplify}

$$\forall\, Stopwatch;\; Chart_{SW};\; Chart'_{SW};\; \bullet$$
$$R \wedge c'_{PP} = cPaused \wedge c'_{Tick} = t\,0 \Rightarrow$$
$$t^{\sim}\, c'_{Tick} = 0 \wedge c'_{PP} = cPaused$$

$\equiv$ {logic and definitions}

$$\mathbf{true}$$

Proofs for the remaining operations can be found in appendix L. Additionally we provide the operation schemas for a further μ-Chart visualisation in appendix M.

8.9 Summary

In this thesis we have added a new operator to the μ-chart semantics. This operator allows us to create schemas that describe how the μ-chart responds to operations defined in Z specifications. By building these schemas we can find refinement relations between the specification and the μ-chart. We are using the μ-charts to visualise Z specifications and by finding a refinement relation we can ensure these visualisations are sound.

Although we have constructed the operation operator to help ease the discovery of refinement relations this is not the only possible use. If you are not experienced with μ-charts, or are investigating particularly complex μ-charts with decomposition and feedback, you may refer to the semantics of the μ-chart to completely understand the meaning and ensure that nothing has been missed. Separating the chart into its component operations is useful for data refinement and for understanding the meaning of the μ-chart piece by piece.

Chapter 9

Animated Visualisations

So far we have only given examples of static visualisations like state diagrams and μ-charts. However, there is another category of visualisations that we will call animated visualisations. Animated visualisation can include moving parts, user interaction, and, most importantly for our investigation, hidden information that is only shown to the user after the visualisation changes in some way. This includes changing the state of the visualisation by simulating an operation being used, getting more details by clicking on a state, or viewing different operations separately from one another.

Specifications that include observation values changing over time are well suited to being visualised using an animated visualisation. Similarly, any operations that are intended to be uncontrollable by the user (e.g. *Tick*) can be animated to create an interactive environment for the user to explore.

We show in this section that refinement can be used to characterise the soundness of animated visualisations.

9.1 Aesthetic Animated Visualisations

Note again that we are not focusing on the aesthetics of animations. The movement of the visualisations is not relevant to the formal meaning of the specification. For example, a visualisation may show an animation of an operation being performed to change the state. If the intermediary frames of this animation are designed to show the transition from one thing to another in a visually pleasing way then we will only consider the before and afterstate of the animation.

For example, consider an animated visualisation of the birthday book specification that shows an image of an open book. When a new name is added to the book an animation plays, hand-drawing the name and date. If the name 'Alan' is added to the book we would consider this to be a single use of the *AddFriend* operation being visualised and not the

letter 'A' being added, then edited to 'Al', 'Ala', and '$Alan$' in order.

Our definitions of animated visualisations and soundness become simpler when we do not need to consider intermediary frames. Additionally, this helps us provide examples in this thesis, as we can show the before and afterstates of an animated transition and describe the animation that takes place in the animated version.

9.2 Simulation

We will mostly be focusing on simulations where the programmer or computer walks the user through the specification step by step. Let us consider a simple simulation of a specification. The visualisation shows the current values of the specification observations and provides a list of possible operations the user can use. When the user uses an operation values change. For an animated visualisation to be sound everything that it shows is possible is also possible in the specification. Additionally, every operation that is enabled in the specification must be usable in the corresponding visualisation state. Figure 9.1 is a visualisation of the Jars example. The user initialises the system, fills $j3$, and transfers the liquid to $j5$. We can see the enabled operations on the right and the values of the state observations on the left for each step of the simulation. It is easy to identify states for this type of visualisation and to see the operations that are enabled in each state.

9.3 Animated Visualisation Definition

Firstly, we give a general definition of what makes a visualisation animated as opposed to static. Because we do not want to restrict our definition to a particular type of animated visualisation we start by providing one major difference and show how this affects our previous definitions of usability and soundness.

Static visualisations show all their information in a single unchanging image while animated visualisations do not. An animation change automatically, because of operations being used, or because of user interaction. Therefore, the information the visualisation has communicated to the user is limited by how much of the visualisation the user has explored.

Because we are visualising Z specifications the soundness of the visualisation depends on how the visualisation shows operations changing the state of the specification. It is important to know which operations are being used and how the states in the visualisation relate to states in the specification. For consistency, and to allow us to easily use Z refinement, we define this information using Z. Our formalisation of Z animations is similar to our formalisation for static visualisations. That is, we still use state and operation schemas to specify what is shown in the visualisation. Firstly, if the animation has a formal visual semantics then we tranform this into Z in order to use Z refinement. Alternatively, if the

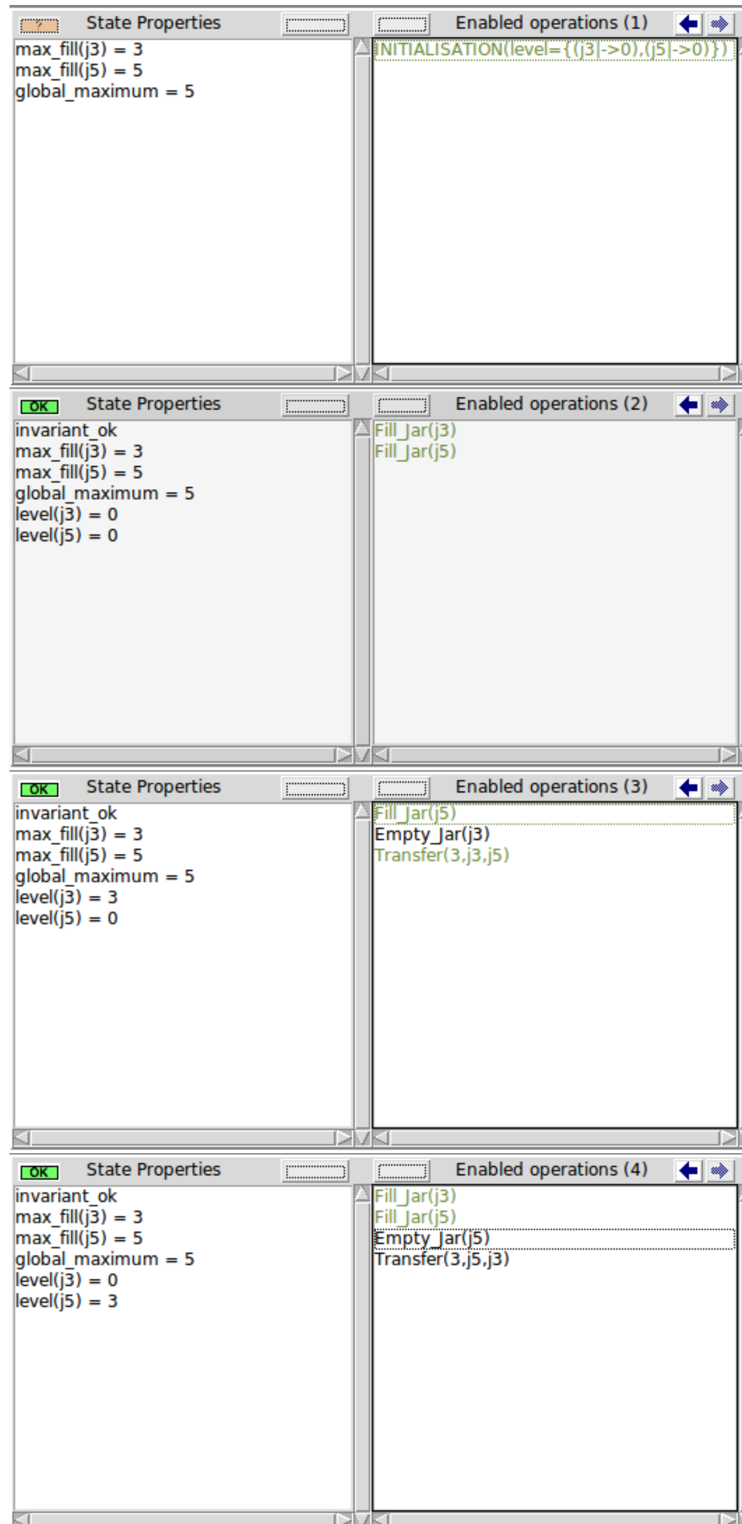


Figure 9.1: ProZ Jars Simulation

animation shows the state observation values, like in Figure 9.1, then we can simply use the abstract state schema observations. Finally, if the state space has changed or is unclear then we can use the given set $[STATES]$ as we did for static visualisations. This set contains all states in the animation and is used to create the following schemas:

$$CStates \mathrel{\widehat=} [c : STATES]$$
$$State_1 \mathrel{\widehat=} [CStates \mid c = State1]$$
$$\ldots$$
$$State_N \mathrel{\widehat=} [CStates \mid c = StateN]$$

These state schemas can then be used to create operation schemas using schema calculus:

$$Transition_{1,2} \mathrel{\widehat=} [\Delta CStates \mid State_1 \wedge State_2']$$

A transition schema $Transition_{i,j}$ is created when the animation changes because of an operation being used. The exact naming convention is not important although each schema must have a unique name. Short names are generally preferred but to avoid multiple transition schemas with the same name we can include the operation name and input values. The following example is from Figure 9.1 and so there are two differences compared to the previous generic example. Firstly, it uses the abstract state space instead of *CStates*. Secondly, it includes the input observation $j?$.

$$Fill1 \mathrel{\widehat=} [\Delta Level;\ j? Jars \mid level = \{(j3, 0), (j5, 0)\} \wedge j? = j3 \wedge level' = \{(j3, 3), (j5, 0)\}]$$

After creating schemas for each transition shown in the animation we combine transitions with the same operation. For example:

$$CFill \mathrel{\widehat=} Fill1 \vee Fill2 \vee \ldots$$

This produces an operation schema for the animation that is conformal with the specification.

9.4 Refinement

As we have in previous chapters, we will be using Z refinement to characterise the soundness of visualisations; in this case however the visualisations are animated. The formalisations for static and animated visualisation produce similar state and operation schemas. This means that we do not need to create a different method for finding a refinement relation for animations.

For large animated visualisations it is unlikely the user will explore the entire visualisation. Therefore, we use our definition for restricted visualisations from chapter 7 to ensure that we only check that what they have explored is sound. In section 7.11 we presented our version of operation refinement that included restrictions:

$$V \sqsubseteq_{W\bullet} U_R =_{df} \overset{\bullet}{V} \subseteq (\overset{\bullet}{U_R})$$

We will use this to show the the formalisation of the animation in Figure 9.1 is sound. The states that this animation has explored are $\langle level = \{(j3, 0), (j5, 0)\}\rangle$, $\langle level = \{(j3, 3), (j5, 0)\}\rangle$, and $\langle level = \{(j3, 0), (j5, 3)\}\rangle$. This is what we use as our restriction. Bindings that do not begin in these states will be totalised in $(\overset{\bullet}{U_R})$. The animation does not contain any transitions that start outside of these three states. However, the lifted animation does as it includes the $\uplus$ and $\perp$ states. The $\subseteq$ relationship holds for these transitions because of the totalisation. For these three states we check that the animation shows transition for each operation and input. For example the first state $\langle level = \{(j3, 0), (j5, 0)\}\rangle$ shows that there is an enabled operation for $Fill_Jar(j3)$ and $Fill_Jar(j5)$ but none of the other operations. In this situation we need to show that this state and input is outside the precondition for the abstract operations. For $Empty_Jar$ this is because $level(j?) > 0$ and for $Transfer$ $amount? > 0$. In this example the animation has enabled operations for every input within that operation's precondition. Next we check that each enabled operation in the animation matches one in the specification. There are nine of these, which we can break up into categories. Two are operations that the user has chosen to use. Four end in an unexplored state so we check that these four also end outside the restriction in the specification. The remaining three are coloured black and would put the simulation into a previously explored state. For the first two operations it is easy to see that they are also in the specification. For example: $\langle level = \{(j3, 0), (j5, 0)\}, j? = 3, level' = \{(j3, 3), (j5, 0)\}\rangle$ exists in both the animation and $Fill_Jar$. It is also easy to find a binding for each of the operations that end outside the restriction. For example: $\langle level = \{(j3, 0), (j5, 0)\}, j? = 5, level' = \{(j3, 0), (j5, 5)\}\rangle$ matches $Fill_Jar(j5)$ as $\{(j3, 0), (j5, 5)\}$ is outside the restriction. For the black operations we need to find a matching binding that ends inside the restriction. For example: $\langle level = \{(j3, 3), (j5, 0)\}, j? = 3, level' = \{(j3, 0), (j5, 0)\}\rangle$ matches $Empty_Jar(j3)$ as $\{(j3, 0), (j5, 0)\}$ is inside the restriction. This shows that

$$\overset{\bullet}{V} \subseteq (\overset{\bullet}{U_R})$$

and so this visualisation is sound.

Note that the black operations have been treated differently. Previously we did not have a distinction between unused operations that ended inside vs. outside the restriction. If we treated this simulation like our previous visualisations and did not change the restriction then the formalisation of this visualisation is actually unsound. This is because it shows that using $Empty_Jar(j3)$ results in a state that is not $\langle level = \{(j3, 0), (j5, 0)\}\rangle$, $\langle level = \{(j3, 3), (j5, 0)\}\rangle$, or $\langle level = \{(j3, 0), (j5, 3)\}\rangle$. However, in the specification this results in $\langle level = \{(j3, 0), (j5, 0)\}\rangle$ so the formalisation of this animation and the specification show different behaviour. This means it would be unsound. So, if the user did not understand the meaning of the black operations the partial animation shown in Figure 9.1 would be misleading. In the proof above we have made suitable changes that ensure that black

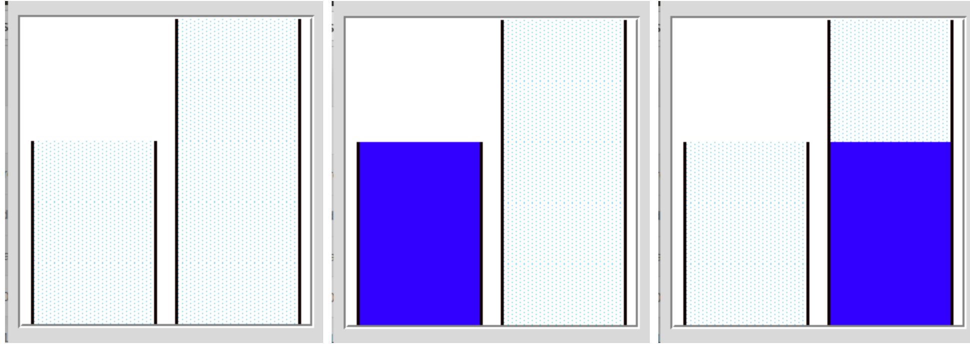


Figure 9.2: ProZ Jars Graphical Animation

operations must end inside the restriction.

9.4.1 Definition by Construction

We can characterise the soundness of a animation that is built using a graphical animation function. Three slides of such a visualisation can be seen in Figure 9.2. This function maps the values of the specification observations to a grid of images. When this visualisation is simulated the image the user sees updates based on the current values of the state observations. The graphical animation function of the Jars example can be found in section 5.5.

So far we have been finding refinement relations using the operations and state space of both the specification and the visualisation. When using data refinement we also needed a retrieve relation to relate the two different state spaces. However, another method allows us to calculate refinement without requiring the operations of the visualisation [29]. The idea is to modify the abstract operations using the retrieve relation resulting in new concrete operations. The new concrete specification will be a refinement of the abstract specification.

We can see how this method can be used in this example. We have the abstract specification, the possible images are our concrete state space, and *ProZ_Settings* relates the abstract and concrete state spaces. We can use this information to calculate a sound visualisation. This method constructs a set of Z operation schemas that operate over a grid of images. Note that the visualisation that we construct is not equal to the ProZ graphical animation. The ProZ graphical animation still has the same limitations as the ProZ simulation.

Not all retrieve relations will result in a sound visualisation. However, if the relation is a function from concrete to abstract then the calculation is simple:

$$CInit \mathrel{\widehat=} \exists\, AState' \bullet AInit \land R'$$

$$COp \mathrel{\widehat=} \exists\, \Delta State_A \bullet R \land AOp \land R'$$

When every state in the specification is mapped to a unique image in the visualisation the relation is bijective and we can use the above calculation. This is the case for the example

in Figure 9.2, so we know that we can calculate a sound visualisation with images like these. Knowing how the visualisation is defined lets us avoid needing to explore a large animated visualisation to find errors and unsound behaviour that could be hidden in the visualisations we have generated.

If we want multiple specification states to map to the same image then the retrieve relation will not be functional. More complex calculations would be needed and it is not guaranteed that a sound visualisation would exist.

We would like to generate a sound animation like the one in Figure 9.2. We start with the following retrieve relation. This relation is functional from concrete to abstract. Each state in the animation represents just one state in the specification.

$$
\begin{array}{|l}
\multicolumn{1}{l}{R} \\
\hline
Level \\
CState \\
\hline
level = \{(j3,0),(j5,0)\} \land State1 \lor \\
level = \{(j3,3),(j5,0)\} \land State2 \lor \\
level = \{(j3,0),(j5,3)\} \land State3 \\
\hline
\end{array}
$$

This function does not provide any information about how $State1, State2$, etc. should be drawn. What is important for this example is that $State1$ should clearly show that both jars are empty. R only includes three states however this schema could be extended to include more. We can then use R to generate concrete operation schema.

$$COp \mathrel{\widehat=} \exists \Delta State_A \bullet R \land AOp \land R'$$

$$CFill \mathrel{\widehat=} \exists \Delta Level \bullet R \land Fill_Jar \land R'$$

$\equiv$ (Expand R')

$$
\begin{aligned}
& CFill \mathrel{\widehat=} [\Delta CState;\ j? \mid \exists \Delta Level \bullet R \land (\\
& level' = \{(j3,0),(j5,0)\} \land State1' \land level(j?) < max_fill(j?) \land \\
& level' = level \oplus \{j? \mapsto max_fill(j?)\} \lor \\
& level' = \{(j3,3),(j5,0)\} \land State2' \land level(j?) < max_fill(j?) \land \\
& level' = level \oplus \{j? \mapsto max_fill(j?)\} \lor \\
& level' = \{(j3,0),(j5,3)\} \land State3' \land level(j?) < max_fill(j?) \land \\
& level' = level \oplus \{j? \mapsto max_fill(j?)\}])
\end{aligned}
$$

$\equiv$ (One-point Rule)

$$CFill \mathrel{\widehat=} [\Delta CState;\ j? \mid \exists\, level \bullet R \land ($$
$$State1' \land level(j?) < max_fill(j?) \land \{(j3,0),(j5,0)\} = level \oplus \{j? \mapsto max_fill(j?)\} \lor$$
$$State2' \land level(j?) < max_fill(j?) \land \{(j3,3),(j5,0)\} = level \oplus \{j? \mapsto max_fill(j?)\} \lor$$
$$State3' \land level(j?) < max_fill(j?) \land \{(j3,0),(j5,3)\} = level \oplus \{j? \mapsto max_fill(j?)\})]$$

$\equiv$ (Expand R and simplify)

$$CFill \mathrel{\widehat=} [\Delta CState;\ j? \mid \exists\, level \bullet$$
$$level = \{(j3,0),(j5,0)\} \land State1 \land State2' \land level(j?) < max_fill(j?) \land$$
$$\{(j3,3),(j5,0)\} = level \oplus \{j? \mapsto max_fill(j?)\}]$$

$\equiv$ (One-point Rule)

$$CFill \mathrel{\widehat=} [\Delta CState;\ j? \mid State1 \land State2' \land \{(j3,0),(j5,0)\}.j? < max_fill(j?) \land$$
$$\{(j3,3),(j5,0)\} = \{(j3,0),(j5,0)\} \oplus \{j? \mapsto max_fill(j?)\}]$$

$\equiv$ (Simplify)

$$CFill \mathrel{\widehat=} [\Delta CState;\ j? \mid State1 \land State2' \land j? = j3]$$

This has generated a simple operation schema for *Fill*. Similarly, we can generate the remaining operations and *CInit* schema.

$$CEmpty \mathrel{\widehat=} [\Delta CState;\ j? \mid State2 \land State1' \land j? = j3 \lor$$
$$State3 \land State1' \land j? = j5]$$

$$CTransfer \mathrel{\widehat=} [\Delta CState;\ j1?;\ j2? amount? \mid amount? = 3 \land State2 \land$$
$$State3' \land j1? = j3 \land j2? = j5 \lor$$
$$amount? = 3 \land State3 \land State2' \land j1? = j5 \land j2? = j3]$$

$$CInit \mathrel{\widehat=} [CState' \mid State1']$$

A refinement relation exists between the specification and the concrete schema that we have generated. If this specification is transformed into a state diagram or animation it is guaranteed to be sound. While this example does not describe what the visualisation looks like, an animation function does. An example of how we can construct a sound animation using an animation function can be found in appendix N.

9.5 Summary

Animated and interactive visualisations are a useful tool and can help demonstrate properties that would be difficult to show statically. However, we are interested in characterising

soundness and animation can be safely removed without affecting the soundness of the visualisation. This allows us to convert animated visualisations into state diagrams and use our existing methods to characterise soundness. We also presented a method that can be used to generate sound visualisations using a graphical animation function.

Chapter 10

Conclusion

In this thesis we have covered a variety of ways to visualise Z specifications and shown how refinement methods can be used to characterise the soundness of these visualisations. In this chapter we summarise the main contributions of this work and discuss future work that can be done in the same area.

The goal of this work was to find a way to formally characterise the soundness of Z specification visualisations. Although visualisations can assist in understanding the complexities of specifications, unsound visualisations can also cause misunderstandings. By finding a way to characterise soundness we can identify unsound visualisations. These visualisations can then be discarded or repaired before being used to validate the Z specifications. By doing this we hope to improve the confidence users have in visualisations and also specifications themselves. Alternatively, this method can be applied to visualisations that are suspect, that have shown unexpected errors, or caused confusion in some way. This helps save time and resources however this will not help avoid problems caused by visualisations that reinforce a misunderstanding of the specification.

To help achieve this goal we used Z refinement. This was because of the underlying meanings of refinement. Originally it was intended that the visualisation of the specification be the abstract version. Clearly, a visualisation is an abstraction of a model however our research did not lead to a characterisation of soundness as we hoped. Instead it suggested a way to begin system development not with specifications, but with more informal visualisations of the imagined system that could then be refined into Z.

Rather than using Z refinement we could have created a custom refinement calculus used to compare specifications and visualisations. This is not unusual as there are a variety of refinement methods that are used to compare different formal languages. This was not done for two main reasons. Firstly, the goal was to validate and help the user be more confident that their visualisation is sound. Z refinement is well-known and has long been used to help verify the applicability and correctness of a refined specification. Secondly, we had

begun investigating a variety of visualisations from simple state diagrams to μ-charts and animated, interactive visualisations. Creating a refinement calculus for "visualisations" in general did not seem feasible.

Instead, we developed methods for converting the visualisations into Z. This achieved two goals. Firstly, having a unified language between the specifications and any visualisations allowed us to make easier comparisons and use the existing Z refinement methods. Secondly, it formalises the visualisation. Visualisations are informal by nature as they have been simplified and abstracted to help the user easily understand the complex specification. However, valuable information about the specification still exists within the visualisation (clearly, as they would otherwise be useless) and it is this information that we retrieve when formalising. If the visualisation is too simple and does not contain sufficient information then it will be unsound.

We described multiple ways to formalise the visualisation using Z, including converting state diagrams into bindings and schemas, and using the underlying Z semantics of μ-charts. Additionally, when an unambiguous formal definition did not exist we showed how the users understanding of the visualisation could be used formally. This method can result in multiple users interpreting the visualisation differently due to their knowledge of the system etc. However, we argue that this method can be more useful in practise than using the underlying formal semantics as the user can check that their *understanding* is sound.

At this point, we could formalise the visualisation and find refinement relations between it and the specification. However, we discovered visualisations that should intuitively be sound that were not a refinement of the specification. This meant that refinement did not characterise the soundness for some visualisation types. The visualisations that were being found as unsound were what we later called restricted visualisations. This type of visualisation could be found in both our handmade examples and the state diagrams generated using simulation. Our definition of restricted visualisations originally included an "unexplored state". The unexplored state was included when the simulation did not cover the entire state space of the specification. Later, this definition was expanded to include visualisations that did not include every specification operation, focused on particular input values, or only visualised a particular selection of transitions. What these visualisations had in common was that the specification being visualised was restricted in some way.

All restricted visualisations were characterised as unsound as they were not a valid refinement of the specification. In chapter 7 we identified this problem and updated our methods to ensure that sound restricted visualisations were actually sound. Our definition of soundness was updated to acknowledge that not all visualisations visualise the entire specification. We also introduced a restriction schema similar to the retrieve relation schema. This restriction schema was then included when finding a refinement relation.

We also researched further into μ-Chart visualisations. This let us show that refinement could characterise the soundness of a wider range of visualisation types. Additionally, the μ-Chart semantics is given in Z. This gives μ-Chart visualisations a formal and unambiguous meaning compared to the more informal visualisation types we had investigated originally. We made some changes to the μ-Chart semantics, most notably the inclusion of the operation operator that converted the single system tick operation schema into a set of operations. When the tick operation schema is split into operations conformal to the specification operations we can easily apply Z refinement.

In chapter 9 we briefly discussed animated visualisations. Unfortunately, we did not find any particularly interesting problems with these visualisations. We concluded that once properties that did not affect soundness, such as aesthetics, were removed the visualisation could be formalised using methods very similar to the methods we used for state diagrams. So, refinement could be used to characterise the soundness of animated visualisations without any changes. The process of formalising the animated visualisations removed a number of interesting properties and it may be that we have removed properties that could have affected the soundness of the visualisation. This is a topic for future work. For example, does the user interacting with the visualisation affect its soundness? We decided that it did not, so long as the user can only interact in ways allowed by the original specification. Future research could begin by investigating if interaction like this does have an effect. Progressing to allow the user total freedom in how they interact with the visualisation and updating how soundness is characterised appropriately.

In the appendices we include examples of the visualisations being converted into Z and apply refinement rules. We also clarify some details that were not suitable for discussion in the main body of the thesis.

10.1 Limitations to our Approach

10.1.1 Scalability

The visualisations and specifications given in the thesis have been small, proof-of-concept examples. This has allowed us to provide step-by-step proofs in the appendices and to focus on the main problem of soundness. However, how well would it scale in practice? Clearly, larger specifications and visualisations would require larger proofs.

As a handmade visualisation becomes larger and more complex it is easier for it to become unsound and more difficult to see the problem. A computer generated visualisation can create extremely large visualisations and proving the soundness can be time consuming if every state and transition needs to be checked. Similarly, as the specification becomes more complex the proofs become more difficult.

Using refinement to characterise the soundness of large visualisations of complex specifications can still be done. Proof assistants and theorem provers can be used to help reduce or automate some of the complexities involved [7, 20, 37].

However, before starting a long and difficult proof that shows that your enormously complex visualisation is actually sound, instead see if you can scale the visualisation back down. Visualisation design best practices state that the visualisation should be clear and easy to understand. The second of Edward Tufte's graphical principles is that "Graphical excellence consists of complex ideas communicated with clarity, precision, and efficiency." [107]. Your complex visualisation might be sound but still unusable due to graphic design problems or mental overload. Because of this, we have included examples that use techniques like grouping together states and syntactic sugar. These small, simple examples visualise a large state space while making it easier for the user to understand and prove the soundness. Similarly, we have shown how complex specifications can be handled through the use of restrictions. Rather than trying to visualise the entire specification using one complex visualisation the specification can be broken down into smaller parts each focusing on a particular problem or idea. This point can be further emphasised using the following thought experiment. Consider that the birthday book specification was only a small part of a much larger social network specification. Rather than creating state diagrams that try to visualisation the entire social network we have scaled it down to examples that are easier to understand and easier to find refinement relations.

10.1.2 Practical Use

Formally proving that the properties of refinement hold is a difficult and time consuming task. This limits the usefulness of this technique until these properties become trivial to prove. We have seen some examples where these properties hold trivially, such as when the visualisation is isomorphic to the specification. Because of this the best way to make this technique useful is to develop tools that support proving the soundness of visualisations.

Our method characterises the soundness of the formalisation of the visualisation. When the formalisation is clear then we use shorthand and directly say that it characterises the soundness of the visualisation. This is the case when we can formalise the visualisation using its semantics or when users have a common understanding of the meaning of the visualisation. Additionally, in section 5.3 we mentioned that different users can interpret the same visualisation in different ways. This would mean that different users would create different formalisations and we would instead say that we characterise the soundness of the user's understanding of the specification. However, non-formal methods users are unable to create a formalisation of the visualisation. This means that we cannot rely solely on this method to ensure that a visualisation is sound for any user. We still need to ensure that

our visualisations are clear, unambiguous, and understandable.

We have focused on Z specifications and different versions of Z refinement. We chose to use Z because of its popularity and we were most familiar with this language. We believe that the underlying principles of our method can be applied to other formal languages. However, until further research is done it is limited to Z specifications.

10.2 Future Work

The work done here focuses exclusively on Z specifications. Future work can be done to show that refinement (or other methods) can be used to characterise the soundness of any visualisation without being limited to just visualisations of Z specifications. In order for this work to be truly useful future research would need to show that it is feasible to apply these methods to visualisations of industrial specifications. If both visualisations and refinement are being used as part of system development then it is possible to use the methods introduced in this thesis to show that the visualisations are sound.

10.2.1 Tool Support

There are a number of steps where additional tool support could make proving the soundness of a visualisation easier. For example, automatically converting a visualisation into schema form. This would be done using the formalisation of the visualisation as well as the observations from the original specification. This tool could be integrated into existing visualisation software or included as part of a tool developed specifically for creating visualisations like the ones shown in this thesis. Other functionality that could be added to this tool includes:

1. The ability to add restrictions to the visualisations based on Chapter 7.
2. In Section 9.4.1 we looked at functional retrieve relations that specify which specification state each visualisation state is associated with. By using these retrieve relations sound visualisations can be constructed automatically.
3. A means to check the proof, both to verify correctness and discover why a proof of soundness could not be found.
4. A method to convert the state diagram into a domain specific visualisation by adding alternative graphics and interactivity.
5. A way to automatically adjust a visualisation when the original specification is refined. This includes removing the appropriate nondetermistic transitions and changing the outgoing transitions when an operation has new enabled states. Additionally, outdated visualisations can be clearly labelled to ensure it is clear why they are no longer sound visualisations of the latest refinement step.

Outside of this visualisation tool, future work also includes more integration with proof assistants. This can include setting up the environment by importing the specification, visualisation, and the proof obligations. In the future, if the proof is able to be automated then the value of this work will greatly increase.

10.2.2 User Studies

A topic for future work is studying existing visualisations, particularly those that have been used in industry, based on this method. How many unsound visualisations can be found, and why were they unsound? It would be useful to characterise different types of errors that can be found in unsound visualisations. This could be used to report common errors and how they can be identified and fixed.

There are multiple user studies that can be performed to learn more about the soundness of visualisations. In section 5.1 we identified a range of potential users, from those creating the specifications to those that only see the visualisations. Some interesting studies include:

1. Are users able to identify unsound visualisations? There are a number of variables that can be changed, including the complexity of the specification and visualisation, different types of visualisation, and different reasons why the visualisation is unsound. This study will allow us to learn what types of errors are noticeable and unnoticeable and so develop a method for identifying problematic errors.

2. How useful is an unsound visualisation? The users are provided a list of questions about a specification. To help answer these questions they are provided with some of the following material. The specification itself, an unsound visualisation with minor errors, a different unsound visualisation with major errors, a sound visualisation, and a detailed explanation of the system being specified. User groups that have received different sets can then be compared based on how the questions have been answered, time taken, confidence level, etc.

Lastly, it would be interesting to formally characterise other visualisation properties such as aesthetics or usability.

Bibliography

[1] Islam Abdelhalim, Steve Schneider, and Helen Treharne. "An Optimization Approach for Effective Formalized fUML Model Checking". In: *Software Engineering and Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 248–262.

[2] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[3] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[4] W Richards Adrion, Martha A Branstad, and John C Cherniavsky. "Validation, verification, and testing of computer software". In: *ACM Computing Surveys (CSUR)* 14.2 (1982), pp. 159–192.

[5] Sten Agerholm and Peter Gorm Larsen. "A lightweight approach to formal methods". In: *International Workshop on Current Trends in Applied Formal Methods*. Springer. 1998, pp. 168–183.

[6] Carina Andersson and Per Runeson. "Verification and validation in industry-a qualitative survey on the state of practice". In: *Proceedings International Symposium on Empirical Software Engineering*. IEEE. 2002, pp. 37–47.

[7] Rob D Arthan. "On formal specification of a proof tool". In: *International Symposium of VDM Europe*. Springer. 1991, pp. 356–370.

[8] Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice*. Prentice-Hall, Inc., 1995.

[9] Yaman Barlas. "Formal aspects of model validity and validation in system dynamics". In: *System Dynamics Review: The Journal of the System Dynamics Society* 12.3 (1996), pp. 183–210.

[10] Yaman Barlas and Stanley Carpenter. "Philosophical roots of model validation: two paradigms". In: *System Dynamics Review* 6.2 (1990), pp. 148–166.

[11] Leonor M Barroca and John A. McDermid. "Formal methods: Use and relevance for the development of safety-critical systems". In: *The Computer Journal* 35.6 (1992), pp. 579–599.

[12] Jacques Bertin. *Graphics and graphic information processing*. Walter de Gruyter, 2011.

[13] Dines Bjørner. "The Vienna Development Method (VDM)". In: *Mathematical Studies of Information Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 326–359.

[14] Dines Bjørner and Cliff B Jones. *The Vienna Development Method: The Meta-Language*. Vol. 61. Springer, 1978.

[15] Eerke Albert Boiten and John Derrick. "IO - refinement in Z". In: *3rd Northern Formal MethodsWorkshop, 1998*. Electronic Workshops in Computing. Springer Verlag, 1998.

[16] Christie Bolton. "Using the Alloy analyzer to verify data refinement in Z". In: *Electronic Notes in Theoretical Computer Science* 137.2 (2005), pp. 23–44.

[17] George S Boolos, John P Burgess, and Richard C Jeffrey. *Computability and logic*. Cambridge University Press, 2002.

[18] Judy Bowen, Steve Jones, and Steve Reeves. "Creating Visualisations of Formal Models of Interactive Medical Devices". In: *Pre-proceedings of Second International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2013)*. 2013, pp. 259–263.

[19] Judy Bowen and Steve Reeves. "A Simplified Z Semantics for Presentation Interaction Models". In: *International Symposium on Formal Methods*. Springer. 2014, pp. 148–162.

[20] Achim D Brucker, Frank Rittinger, and Burkhart Wolff. "HOL-Z 2.0". In: *Journal of Universal Computer Science* 9.2 (2003), pp. 152–172.

[21] Jordi Cabot and Martin Gogolla. "Object constraint language (OCL): a definitive guide". In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer. 2012, pp. 58–90.

[22] Mackinlay Card. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[23] Stuart K Card, George G Robertson, and Jock D Mackinlay. "The information visualizer, an information workspace". In: *Proceedings of the SIGCHI Conference on Human factors in computing systems*. ACM. 1991, pp. 181–186.

[24] Guido de Caso et al. "Automated abstractions for contract validation". In: *IEEE Transactions on Software Engineering* 38.1 (2010), pp. 141–162.

[25] Edmund M Clarke and Jeannette M Wing. "Formal methods: State of the art and future directions". In: *ACM Computing Surveys (CSUR)* 28.4 (1996), pp. 626–643.

[26] William S Cleveland. *Visualizing data*. Hobart Press, 1993.

[27] Darren Cofer and Steven P Miller. *Formal methods case studies for DO-333*. NASA, 2014.

[28] Michelle L Crane and Juergen Dingel. "On the semantics of UML state machines: Categorization and comparision". In: *Technical Report 2005-501, School of Computing, Queen's*. Citeseer. 2005.

[29] John Derrick and Eerke A Boiten. *Refinement in Z and Object-Z: foundations and advanced applications*. Springer Science & Business Media, 2013.

[30] John Derrick and Heike Wehrheim. "Using coupled simulations in non-atomic refinement". In: *International Conference of B and Z Users*. Springer. 2003, pp. 127–147.

[31] Brian Dobing and Jeffrey Parsons. "How UML is used". In: *Communications of the ACM* 49.5 (2006), pp. 109–113.

[32] Nicolas Dulac et al. "On the use of visualization in formal requirements specification". In: *Proceedings IEEE Joint International Conference on Requirements Engineering*. IEEE. 2002, pp. 71–80.

[33] Aaron M Dutle et al. "Software validation via model animation". In: *International Conference on Tests and Proofs*. Springer. 2015, pp. 92–108.

[34] Steve Easterbrook et al. "Experiences using lightweight formal methods for requirements modeling". In: *IEEE Transactions on Software Engineering* 24.1 (1998), pp. 4–14.

[35] Alessandro Fantechi, Francesco Flammini, and Stefania Gnesi. "Formal methods for railway control systems". In: *International Journal on Software Tools for Technology Transfer* 16.6 (Nov. 2014), pp. 643–646.

[36] Stephen Few. *Now you see it: simple visualization techniques for quantitative analysis*. Analytics Press, 2009.

[37] Leonardo Freitas. *Proving theorems with Z/Eves*. URL: `https://www.cs.york.ac.uk/ftpdir/pub/leo/mefes/zeves/tutorials/CRG-3.pdf` (visited on 04/30/2018).

[38] Ana Garis, Alcino Cunha, and Daniel Riesco. "Translating Alloy specifications to UML class diagrams annotated with OCL". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2011, pp. 221–236.

[39] Fahad Rafique Golra et al. "Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2018, pp. 54–69.

[40] Niusha Hakimipour, Paul Strooper, and Andy Wellings. "TART: Timed-automata to real-time Java tool". In: *2010 8th IEEE International Conference on Software Engineering and Formal Methods*. IEEE. 2010, pp. 299–309.

[41] Anthony Hall. "Seven myths of formal methods". In: *IEEE software* 7.5 (1990), pp. 11–19.

[42] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[43] Robert L Harris. *Information graphics: A comprehensive illustrated reference*. Oxford University Press, 2000.

[44] Daniel Hazel, Paul Strooper, and Owen Traynor. "POSSUM: An Animator for the SUM Specification Language". In: *Asia-Pacific Software Engineering Conference and International Computer Science Conference*. IEEE Computer Society, 1997, pp. 42–51.

[45] Martin C Henson, Steve Reeves, and Jonathan P Bowen. "Z Logic and its Consequences". In: *Computing and Informatics* 22.3-4 (2012), pp. 381–415.

[46] MA Hewitt, CM O'Halloran, and Chris T Sennett. "Experiences with PiZA, an animator for Z". In: *International Conference of Z Users*. Springer. 1997, pp. 35–51.

[47] Jozef JM Hooman, S Ramesh, and Willem-Paul de Roever. "A compositional axiomatization of Statecharts". In: *Theoretical Computer Science* 101.2 (1992), pp. 289–335.

[48] Bardh Hoxha, Nikolaos Mavridis, and Georgios Fainekos. "VISPEC: A graphical tool for elicitation of MTL requirements". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2015, pp. 3486–3492.

[49] Bardh Hoxha et al. *Towards formal specification visualization for testing and monitoring of cyber-physical systems*. 2014. URL: `https://pdfs.semanticscholar.org/68a4/d2c63a74bb365ba024832bc266e2ca3a6042.pdf` (visited on 09/30/2018).

[50] Edwin Hutchins. *Cognition in the Wild*. MIT press, 1995.

[51] Edwin Hutchins, James D. Hollan, and Donald A. Norman. "Direct Manipulation Interfaces". In: *Human-Computer Interaction* 1 (1985), pp. 311–338.

[52] Akram Idani and Nicolas Stouls. "When a formal model rhymes with a graphical notation". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2014, pp. 54–68.

[53] Juha Itkonen, Mika V Mantyla, and Casper Lassenius. "How do testers do it? An exploratory study on manual testing practices". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2009, pp. 494–497.

[54] Daniel Jackson. "Alloy: a lightweight object modelling notation". In: *ACM Transactions on Software Engineering and Methodology* 11.2 (2002), pp. 256–290.

[55] Daniel Jackson. *Software Abstractions: logic, language, and analysis.* MIT press, 2012.

[56] Jonathan Jacky. *The way of Z: practical programming with formal methods.* Cambridge University Press, 1997.

[57] He Jifeng. "Process simulation and refinement". In: *Formal Aspects of Computing* 1.1 (Mar. 1989), pp. 229–241.

[58] Soon-Kyeong Kim and David Carrington. "Visualization of formal specifications". In: *Proceedings Sixth Asia Pacific Software Engineering Conference.* IEEE. 1999, pp. 102–109.

[59] Eva Kühn and Sophie Therese Radschek. "An initial user study comparing the readability of a graphical coordination model with Event-B notation". In: *International Conference on Software Engineering and Formal Methods.* Springer. 2017, pp. 574–590.

[60] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. "Visualising event-B models with B-motion studio". In: *International Workshop on Formal Methods for Industrial Critical Systems.* Springer. 2009, pp. 202–204.

[61] Lukas Ladenberger, Ivaylo Dobrikov, and Michael Leuschel. "An approach for creating domain specific visualisations of CSP models". In: *International Conference on Software Engineering and Formal Methods.* Springer. 2014, pp. 20–35.

[62] Lukas Ladenberger and Michael Leuschel. "Mastering the visualization of larger state spaces with projection diagrams". In: *International Conference on Formal Engineering Methods.* Springer. 2015, pp. 153–169.

[63] Jill H Larkin and Herbert A Simon. "Why a diagram is (sometimes) worth ten thousand words". In: *Cognitive science* 11.1 (1987), pp. 65–100.

[64] Diego Latella, Istvan Majzik, and Mieke Massink. "Towards a formal operational semantics of UML statechart diagrams". In: *International Conference on Formal Methods for Open Object-Based Distributed Systems.* Springer. 1999, pp. 331–347.

[65] Yves Ledru. "Using Jaza to animate RoZ specifications of UML class diagrams". In: *2006 30th Annual IEEE/NASA Software Engineering Workshop.* IEEE. 2006, pp. 253–262.

[66] Yves Ledru et al. "Validation of security policies by the animation of Z specifications". In: *Proceedings of the 16th ACM symposium on Access control models and technologies.* ACM. 2011, pp. 155–164.

[67] Michael Leuschel and Michael Butler. "ProB: A model checker for B". In: *International Symposium of Formal Methods Europe*. Springer. 2003, pp. 855–874.

[68] Michael Leuschel and Michael Butler. "ProB: an automated analysis toolset for the B method". In: *International Journal on Software Tools for Technology Transfer* 10.2 (2008), pp. 185–203.

[69] Michael Leuschel, Mireille Samia, and Jens Bendisposto. "Easy graphical animation and formula visualisation for teaching B". In: (2008).

[70] Michael Leuschel and Edd Turner. "Visualising larger state spaces in ProB". In: *International Conference of B and Z Users*. Springer. 2005, pp. 6–23.

[71] David Lightfoot. *Formal specification using Z*. Palgrave, 2001.

[72] Shaoying Liu and Hao Wang. "An automated approach to specification animation for validation". In: *Journal of Systems and Software* 80.8 (2007), pp. 1271–1285.

[73] Shaoying Liu et al. "SOFL: A formal engineering methodology for industrial applications". In: *IEEE Transactions on Software Engineering* 24.1 (1998), pp. 24–45.

[74] Petra Malik. "A retrospective on CZT". In: *Software: Practice and Experience* 41.2 (2011), pp. 179–188.

[75] Petra Malik and Mark Utting. "CZT: A framework for Z tools". In: *International Conference of B and Z Users*. Springer. 2005, pp. 65–84.

[76] Aad Mathijssen and A Johannes Pretorius. "Verified design of an automated parking garage". In: *International Workshop on Parallel and Distributed Methods in Verification*. Springer. 2006, pp. 165–180.

[77] Sun Meng, Zhang Naixiao, and Luís Soares Barbosa. "On semantics and refinement of UML statecharts: A coalgebraic view". In: *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.* IEEE. 2004, pp. 164–173.

[78] Tim Miller and Paul Strooper. "A case study in model-based testing of specifications and implementations". In: *Software Testing, Verification and Reliability* 22.1 (2012), pp. 33–63.

[79] Tim Miller and Paul Strooper. "A framework and tool support for the systematic testing of model-based specifications". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 12.4 (2003), pp. 409–439.

[80] Tim Miller and Paul Strooper. "Supporting the software testing process through specification animation". In: *First International Conference onSoftware Engineering and Formal Methods, 2003. Proceedings.* IEEE. 2003, pp. 14–23.

[81] Michael Möller et al. “Integrating a formal method into a software engineering process with UML and Java”. In: *Formal Aspects of Computing* 20.2 (2008), pp. 161–204.

[82] Glenford J Myers et al. *The art of software testing*. Vol. 2. Wiley Online Library, 2004.

[83] Kumiyo Nakakoji, Akio Takashima, and Yasuhiro Yamamoto. “Cognitive effects of animated visualization in exploratory visual data analysis”. In: *Proceedings Fifth International Conference on Information Visualisation*. IEEE. 2001, pp. 77–84.

[84] Don Norman. *Things that make us smart: Defending human attributes in the age of the machine*. Diversion Books, 2014.

[85] Jan Peleska and Wen-ling Huang. “Industrial-strength model-based testing of safety-critical systems”. In: *International Symposium on Formal Methods*. Springer. 2016, pp. 3–22.

[86] Jan Philipps and Peter Scholz. “Compositional specification of embedded systems with statecharts”. In: *Colloquium on Trees in Algebra and Programming*. Springer. 1997, pp. 637–651.

[87] Colin Pilbrow and Steve Reeves. “Characterising Sound Visualisations of Specifications using Micro-charts and Refinement”. In: *24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2017, pp. 612–617.

[88] Colin Pilbrow and Steve Reeves. “Using state machines for the visualisation of specifications via refinement”. In: *Proceedings of the 24th Australasian Software Engineering Conference (ASWEC)*. Vol. 2. ACM. 2015, pp. 106–110.

[89] Daniel Plagge and Michael Leuschel. “Validating Z specifications using the ProB animator and model checker”. In: *International Conference on Integrated Formal Methods*. Springer. 2007, pp. 480–500.

[90] Amir Pnueli and Michal Shalev. “What is in a step: On the semantics of Statecharts”. In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1991, pp. 244–264.

[91] Christophe Ponsard et al. “Early verification and validation of mission critical systems”. In: *Formal Methods in System Design* 30.3 (2007), p. 233.

[92] AJ Pretorius. “Visualization of state transition graphs”. PhD thesis. Eindhoven University of Technology, 2008.

[93] Nafees Qamar, Yves Ledru, and Akram Idani. “Validation of security-design models using Z”. In: *International Conference on Formal Engineering Methods*. Springer. 2011, pp. 259–274.

[94] Bryan Ratcliff. *Introducing Software Engineering Specification Using Z: A Practical Case Study Approach.* McGraw-Hill, Inc., 1994.

[95] Greg Reeve. "A refinement theory for μ-Charts". PhD thesis. The University of Waikato, 2005.

[96] Greg Reeve and Steve Reeves. "μ-Charts and Z: Hows, whys, and wherefores". In: *International Conference on Integrated Formal Methods.* Springer. 2000, pp. 255–276.

[97] Greg Reeve and Steve Reeves. *The syntax and semantics of μ-Charts.* Tech. rep. Department of Computer Science, University of Waikato, 2004.

[98] Wolfgang Reisig. *Petri nets: an introduction.* Vol. 4. Springer Science & Business Media, 2012.

[99] Howard L Resnikoff. *The illusion of reality.* Springer Science & Business Media, 2012.

[100] Hendrik Roehm et al. "Industrial Examples of Formal Specifications for Test Case Generation." In: *ARCH@ CPSWeek.* 2015, pp. 80–88.

[101] Omar Salman. "Animation of Z Specifications by Translation to Prolog". In: *Dogus University Journal* 155.1 (2000), pp. 155–167.

[102] Linda B Sherrell and Doris L Carver. "Experiences in translating Z designs to Haskell implementations". In: *Software: Practice and Experience* 24.12 (1994), pp. 1159–1178.

[103] J Michael Spivey. *The Z notation: a reference manual. International Series in Computer Science.* 1992.

[104] Susan Stepney and Stephen P Lord. "Formal specification of an access control system". In: *Software: Practice and Experience* 17.9 (1987), pp. 575–593.

[105] Harold Thimbleby. "Interaction walkthrough: evaluation of safety critical interactive systems". In: *International Workshop on Design, Specification, and Verification of Interactive Systems.* Springer. 2006, pp. 52–66.

[106] Ulyana Tikhonova, Maarten Manders, and Rimco Boudewijns. "Visualization of formal specifications for understanding and debugging an industrial DSL". In: *Federation of International Conferences on Software Technologies: Applications and Foundations.* Springer. 2016, pp. 179–195.

[107] Edward R Tufte. *The visual display of quantitative information.* Vol. 2. Graphics press Cheshire, CT, 2001.

[108] John W Tukey and Paul A Tukey. "Computer graphics and exploratory data analysis: An introduction". In: *The Collected Works of John W. Tukey: Graphics: 1965-1985* 5 (1988), p. 419.

[109] Mark Utting and Petra Malik. "Unit testing of Z specifications". In: *International Conference on Abstract State Machines, B and Z.* Springer. 2008, pp. 309–322.

[110] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (2012), pp. 297–312.

[111] University of Waikato. *AMuZed and ZooM.* 2004. URL: `https://www.cs.waikato.ac.nz/research/fm/amuzed.html` (visited on 04/30/2018).

[112] Dolores R. Wallace and Roger U. Fujii. "Software verification and validation: an overview". In: *IEEE Software* 6.3 (1989), pp. 10–17.

[113] Niklaus Wirth. "Program development by stepwise refinement". In: *Communications of the ACM* 26.1 (1983), pp. 70–74.

[114] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof.* Vol. 39. Prentice Hall Englewood Cliffs, 1996.

[115] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* 41.4 (2009), p. 19.

[116] J. B. Wordsworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., 1992.

[117] Hong Zhu, Patrick AV Hall, and John HR May. "Software unit test coverage and adequacy". In: *ACM computing surveys (CSUR)* 29.4 (1997), pp. 366–427.

Appendices

The appendices contain proofs for many of the visualisations we have looked at in the thesis. In each case the theorem being proved is that a certain visualisation is or is not sound. Typically, these proofs include a formalisation of the visualisation and we check if the refinement properties hold.

- Appendix A contains the refinement proof obligations and some common rules that will be used throughout the proofs.
- In Appendix B we look at an unsound visualisation of the birthday book specification. We formalise this visualisation using schema calculus and use data refinement.
- In Appendices C and D we discuss how two visualisations can be formalised using bindings and show they are sound using operation refinement.
- In Appendix E we create a schema formalism of a birthday book visualisation using the same state space as the specification. This lets us use operation refinement and find that the visualisation is unsound.
- In Appendix F we investigate a sound visualisation of the birthday book specification using schema calculus and data refinement.
- This is followed by a similar visualisation in Appendix G that includes an unexplored state.
- In Appendix H we look at a partial visualisation that only includes two of the operations of the stopwatch specification. Additionally, we discuss the consequences of this visualisation lacking an initial state.
- In Appendices I and J we look at two partial visualisations and investigate the effect of using different restrictions.
- In Appendix K we look at a partial visualisation with multiple unexplored states and show that it is a sound visualisation of a restricted jars specification.
- In Appendix L we present the complete formalisation of a μ-Chart visualisation that uses composition. After applying the operation operator we use data refinement to show the visualisation is sound.
- In Appendix M we present the final operation schemas of a μ-Chart visualisation that uses a local variable and assignment. This is shown to be sound using data refinement.
- Finally, in Appendix N we construct a sound visualisation using an animation function.

Appendix A

Proof Rules

We described the different proof obligations in chapter 4. Typically, if our specification and visualisation share the same state space then we will use operation refinement from section 4.3:

Applicability property:

$$\forall\, State;\ ?AOp \bullet \mathbf{pre}\, AOp \Rightarrow \mathbf{pre}\, COp$$

Correctness property:

$$\forall\, State;\ State';\ ?AOp;\ !AOp \bullet \mathbf{pre}\, AOp \land COp \Rightarrow AOp$$

Otherwise, we use data refinement:

Initialisation property:

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \land R'$$

Applicability property:

$$\forall\, AState;\ CState;\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$$

Correctness property:

$$\forall\, AState;\ CState;\ CState';\ ?AOp_i;\ !AOp_i \bullet \mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow \exists\, AState' \bullet R' \land AOp_i$$

For each example if all proof obligations are satisfied we have found a refinement relation and conclude that the visualisation is sound. In order to prove that these properties hold we first formalise the visualisation. Additionally, we may need to create a retrieve relation that describes how the states in the specification and visualisation are related. For partial visualisations we also give a restriction that is applied to the specification operation schema.

The *one-point rule* is used to eliminate $\exists$. Where x is not free in t:

$$\exists\, x : S \bullet t \in S \land (P \land x = t) \equiv P[t/x]$$

If we know the value of a variable we can substitute the value for the variable and eliminate it from the existential quantification. This rule will be used in data refinement when proving the correctness property holds.

To eliminate the universal quantification we use two main rules. If we can show that P is true for all values of quantified variable x then $\forall x : X \bullet P$ is true. This is usually done by assuming x is arbitrary, i.e. no assumptions made about it.

$$\forall x : X \bullet P \equiv \mathbf{true} \text{ if } P \equiv \mathbf{true}$$

Alternatively, if we can find even one counterexample such that P is **false** then $\forall x : X \bullet P$ is **false**.

$$\forall x : X \bullet P \equiv \mathbf{false} \text{ if } P[t/x] \equiv \mathbf{false}$$

The majority of the proofs use implication so we use the following rules:

$$\mathbf{false} \Rightarrow P \equiv \mathbf{true}$$
$$P \Rightarrow \mathbf{true} \equiv \mathbf{true}$$
$$\mathbf{true} \Rightarrow \mathbf{false} \equiv \mathbf{false}$$
$$P \Rightarrow P \equiv \mathbf{true}$$
$$(Q \land P) \Rightarrow P \equiv \mathbf{true}$$
$$(P \lor Q) \Rightarrow R \equiv (P \Rightarrow R) \land (Q \Rightarrow R)$$
$$P \Rightarrow (Q \land R) \equiv (P \Rightarrow Q) \land (P \Rightarrow R)$$
$$P \Rightarrow Q \equiv P \Rightarrow Q \land P$$

In section 7.10 we use some natural deduction rules to complete our proofs. While the standard rules can be found elsewhere we also introduce two that are specific to Z [45]. In the following rules U is an operation schema with type T. T can be split into the before and after observations T^{in} and T^{out}. t_0 is a binding with type T^{in} and t_1' is a binding with type T^{out}. These bindings can be concatenated $t_0 \bigstar t_1'$ as the bindings are disjoint. This results in a binding with type T. $y =_{T^{in}} t$ if the before observations and values in bindings y and t are equal.

When y is a fresh binding and P is any proposition we can use the following precondition elimination rule:

$$\frac{\mathbf{pre}\, U\, t \qquad y \in U,\ y =_{T^{in}} t \vdash P}{P}\ \mathbf{pre}-$$

An operation U can be lifted and totalised using the following introduction rule. T^* shows that the type T has been lifted to include $\bot$. If $t_0 \bigstar t_1 \in T$ then $t_0 \bigstar t_1 \in T^*$

$$\frac{t_0 \bigstar t_1 \in T^* \qquad \mathbf{pre}\ U\ t_0 \vdash t_0 \bigstar t_1 \in U}{t_0 \bigstar t_1 \in \overset{\bullet}{U}}\ \bullet +$$

Appendix B

Proofs for Figure 6.1

Figure 6.1 is a visualisation of the birthday book specification that splits the state space into two states; one where the book is empty and one where it is not. We begin by defining the visualisation using schema calculus. We have two state schemas, $State1$ and $State2$, and use these to build the transitions. We also include input observations so that these operations are conformal to the specification operations.

$$Add \mathrel{\widehat=} State2' \land [\, name? : NAME;\ date? : DATE \,]$$
$$Edit \mathrel{\widehat=} State2 \land State2' \land [\, name? : NAME;\ date? : DATE \,]$$
$$Remove \mathrel{\widehat=} State2 \land (State1' \lor State2') \land [\, name? : NAME \,]$$

We then calculate the preconditions of these operations.

$$\mathbf{pre}\, Add = \mathbf{true}$$
$$\mathbf{pre}\, Edit = State2$$
$$\mathbf{pre}\, Remove = State2$$

To use data refinement we need a retrieve relation. This schema matches the states where $birthdaybook = \varnothing$ with $State1$ and the remaining states with $State2$. We include the schema $SDState$ which is the state space of the visualisation. $SDState \mathrel{\widehat=} State1 \oplus State2$

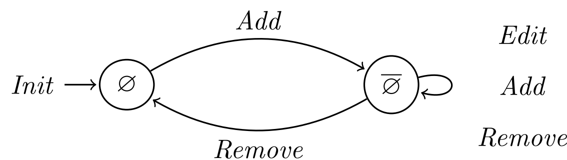


Figure 6.1: Empty or not empty (repeated from page 58)

$$\begin{array}{|l}
\multicolumn{1}{l}{R} \\
\hline
State \\
SDState \\
\hline
birthdaybook = \varnothing \land State1 \lor \\
birthdaybook \neq \varnothing \land State2 \\
\hline
\end{array}$$

We begin by checking the initialisation property.

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \land R'$$

$\equiv$ (Substitution)

$$\forall\, SDState' \bullet SDInit \Rightarrow \exists\, State' \bullet Init \land (birthdaybook' = \varnothing \land State1' \lor birthdaybook' \neq \varnothing \land State2')$$

$\equiv$ (Substitution)

$$\forall\, SDState' \bullet State1' \Rightarrow \exists\, State' \bullet known' = \varnothing \land (birthdaybook' = \varnothing \land State1' \lor birthdaybook' \neq \varnothing \land State2')$$

$\equiv$(One-point rule $known' = \operatorname{dom} birthdaybook'$)

$$\forall\, SDState' \bullet State1' \Rightarrow State1'$$

$\equiv$ $(P \Rightarrow P)$

true

Next, we show the applicability property holds for *Add*:

$$\forall\, AState;\ CState;\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$$

$\equiv$ (Substitution)

$$\forall\, State;\ SDState;\ ?AOp_i \bullet name? \notin known \land R \Rightarrow \mathbf{true}$$

$\equiv$ $(P \Rightarrow \mathbf{true})$

true

Next, we show the applicability property holds for *Edit*:

$$\forall\, AState;\ CState;\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$$

$\equiv$ (Substitution)

$$\forall\, State;\ SDState;\ ?AOp_i \bullet name? \in known \land (birthdaybook = \varnothing \land State1 \lor birthdaybook \neq \varnothing \land State2 \Rightarrow State2)$$

$\equiv$(Distribution, $\varnothing$)

$$\forall State;\ SDState;\ ?AOp_i \bullet name? \in known \wedge birthdaybook \neq \varnothing \wedge State2 \Rightarrow State2)$$

$\equiv$ $(P \wedge Q \Rightarrow P)$

$$\textbf{true}$$

The preconditions for *Edit* are the same as *Remove* so applicability also holds for *Remove*.

Finally, we need to show the correctness property holds. We start we *Add*:

$$\forall AState;\ CState;\ CState';\ ?AOp_i \bullet \textbf{pre}\, AOp_i \wedge R \wedge COp_i \Rightarrow \exists AState' \bullet R' \wedge AOp_i$$

$\equiv$ (Substitution)

$$\forall State;\ SDState;\ SDState';\ ?AOp_i \bullet name? \notin known \wedge R \wedge State2' \Rightarrow$$
$$\exists State' \bullet R' \wedge name? \notin known \wedge birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$$

$\equiv$(One-point rule)

$$\forall State;\ SDState;\ SDState';\ ?AOp_i \bullet name? \notin known \wedge (birthdaybook = \varnothing \wedge State1 \vee$$
$$birthdaybook \neq \varnothing \wedge State2) \wedge State2' \Rightarrow$$
$$(birthdaybook \cup \{name? \mapsto date?\} = \varnothing \wedge State1' \vee$$
$$birthdaybook \cup \{name? \mapsto date?\} \neq \varnothing \wedge State2') \wedge name? \notin known$$

$\equiv$ ($\varnothing$)

$$\forall State;\ SDState;\ SDState';\ ?AOp_i \bullet name? \notin known \wedge (birthdaybook = \varnothing \wedge State1 \vee$$
$$birthdaybook \neq \varnothing \wedge State2) \wedge State2' \Rightarrow$$
$$State2' \wedge name? \notin known$$

$\equiv$ $(P \wedge Q \Rightarrow P)$

$$\textbf{true}$$

Next, we check correctness for *Remove*:

$$\forall AState;\ CState;\ CState';\ ?AOp_i \bullet \textbf{pre}\, AOp_i \wedge R \wedge COp_i \Rightarrow \exists AState' \bullet R' \wedge AOp_i$$

$\equiv$ (Substitution)

$$\forall State;\ SDState;\ SDState';\ ?AOp_i \bullet name? \in known \wedge (birthdaybook = \varnothing \wedge State1 \vee$$
$$birthdaybook \neq \varnothing \wedge State2) \wedge State2 \wedge (State1' \vee State2') \Rightarrow$$
$$\exists State' \bullet (birthdaybook' = \varnothing \wedge State1' \vee birthdaybook' \neq \varnothing \wedge State2') \wedge$$
$$name? \in known \wedge birthdaybook' = name? \lhd birthdaybook$$

$\equiv$(One-point rule)

$$\forall State;\ SDState;\ SDState';\ ?AOp_i \bullet name? \in known \land birthdaybook \neq \varnothing \land$$
$$State2 \land (State1' \lor State2') \Rightarrow$$
$$(name? \lhd birthdaybook = \varnothing \land State1' \lor name? \lhd birthdaybook \neq \varnothing \land State2') \land$$
$$name? \in known$$

Counterexample $State2 \land State2' \land birthdaybook = \{(A, T)\} \land name? = A$

$$\mathbf{true} \land \mathbf{true} \land \mathbf{true} \land \mathbf{true} \Rightarrow$$
$$(A \lhd \{(A, T)\} = \varnothing \land \mathbf{false} \lor A \lhd \{(A, T)\} \neq \varnothing \land \mathbf{true}) \land A \in \{A\}$$

$\equiv$ ($\varnothing$)

$$\mathbf{true} \Rightarrow \mathbf{false}$$

$\equiv$ ($\mathbf{true} \Rightarrow \mathbf{false}$)

$$\mathbf{false}$$

So, the *Remove* operation is not correct. This is because you can remove the last name in the book and stay in *State*2. This means that the visualisation is not sound and should not be used to help validate the *Remove* operation.

Appendix C

Proofs for Figure 6.3

Figure 6.3 is a visualisation of the *Fill_Jar* and *Empty_Jar* operations from the *Jars* specification. This state diagram was defined using the sets of bindings $(Q, \Sigma, \delta, \delta_0)$. We begin by converting the diagram into bindings without removing the variables x_0 and x_1. Although this means the bindings are not valid, this is still a useful intermediary step.

$$Q = \{\langle j3 \mapsto 0, j5 \mapsto 0\rangle, \langle j3 \mapsto x_0\rangle, \langle j3 \mapsto X_0, j5 \mapsto 5\rangle, \langle j3 \mapsto x_0, j5 \mapsto 0\rangle,$$
$$\langle j5 \mapsto x_1\rangle, \langle j3 \mapsto 3, j5 \mapsto x_1\rangle, \langle j3 \mapsto 0, j5 \mapsto x_1\rangle\}$$

We begin with the set of seven states in the diagram. There are two changes we must make to convert this into a set of valid bindings. Firstly, the variables x_0 and x_1 are removed. This is done by replacing the bindings containing these variables with multiple bindings where the variables have been replaced with the possible values. For example, the binding $\langle j3 \mapsto x_0\rangle$ is replaced with $\langle j3 \mapsto 0\rangle$, $\langle j3 \mapsto 1\rangle$, $\langle j3 \mapsto 2\rangle$ and $\langle j3 \mapsto 3\rangle$.

Secondly, the size of the bindings in Q is not consistent, as two states only refer to one of the observations. This is because in these states the value of the omitted observation is not relevant, and can have any valid value. So, similarly to the first step, we replace these bindings with bindings that contain both observations. For example, the binding $\langle j3 \mapsto 0\rangle$ that we added in the previous step is replaced with the bindings $\langle j3 \mapsto 0, j5 \mapsto 0\rangle$,$\langle j3 \mapsto 0, j5 \mapsto 1\rangle$,$\langle j3 \mapsto 0, j5 \mapsto 2\rangle$, and so on. Replacing the state bindings with valid bindings results in 24 bindings, which we will not show in full. Because this type diagram does not have distinct unique states like a state machine, many bindings are duplicates and are removed from the set of bindings Q. For example $\langle j3 \mapsto 0, j5 \mapsto 0\rangle$ can be found in five of the states.

Σ is the set of observations in the state diagram. We are visualising *Init*, *Fill_Jar* and *Empty_Jar*, where *Fill_Jar* and *Empty_Jar* both have input observation $j?$. δ_0 is the initial state $\langle j3 \mapsto 0, j5 \mapsto 0\rangle$.

For the set of transitions δ, we start by writing the four transitions as invalid bindings.

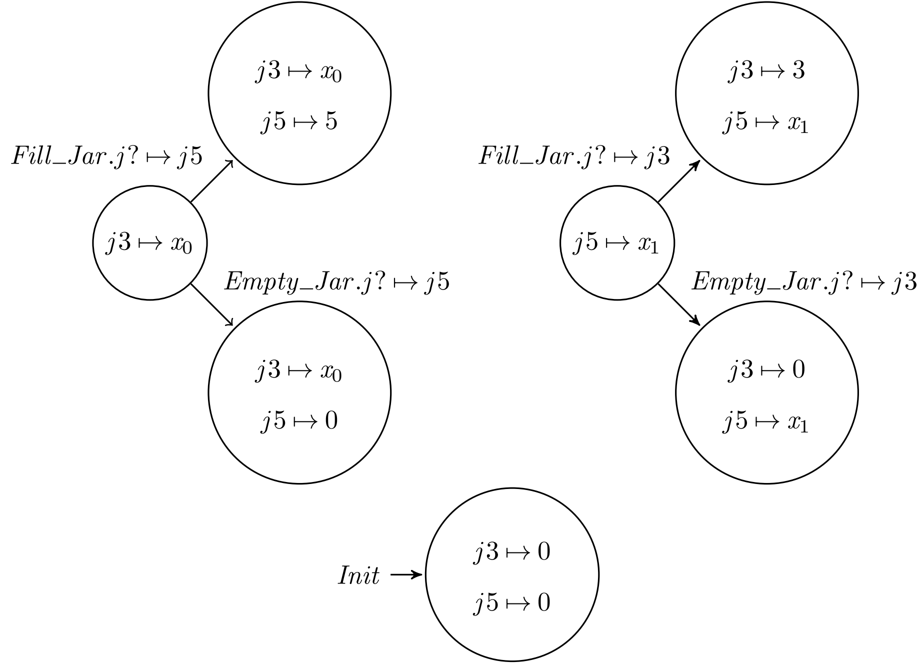


Figure 6.3: State Diagram of Jars (repeated from page 60)

Once again, the variables are present and some bindings are missing observations.

$$\delta = \{\langle j3 \mapsto x_0, j? \mapsto j5, j3' \mapsto x_0, j5' \mapsto 5\rangle, \langle j3 \mapsto x_0, j? \mapsto j5, j3' \mapsto x_0, j5' \mapsto 0\rangle$$
$$\langle j5 \mapsto x_1, j? \mapsto j3, j3' \mapsto 3, j5' \mapsto x_1\rangle \langle j5 \mapsto x_1, j? \mapsto j3, j3' \mapsto 0, j5' \mapsto x_1\rangle\}$$

Note that this representation does not include the operation that is being used. Because this is important, another way of representing δ is by separating the bindings into distinct operations.

$$\delta Fill_Jar = \{\langle j3 \mapsto x_0, j? \mapsto j5, j3' \mapsto x_0, j5' \mapsto 5\rangle,$$
$$\langle j5 \mapsto x_1, j? \mapsto j3, j3' \mapsto 3, j5' \mapsto x_1\rangle\}$$
$$\delta Empty_Jar = \{\langle j3 \mapsto x_0, j? \mapsto j5, j3' \mapsto x_0, j5' \mapsto 0\rangle,$$
$$\langle j5 \mapsto x_1, j? \mapsto j3, j3' \mapsto 0, j5' \mapsto x_1\rangle\}$$

The same steps that we applied to the state bindings are also used here. For example, $\langle j3 \mapsto x_0, j? \mapsto j5, j3' \mapsto x_0, j5' \mapsto 5\rangle$ is replaced with $\langle j3 \mapsto 0, j5 \mapsto 0, j? \mapsto j5, j3' \mapsto 0, j5' \mapsto 5\rangle$,$\langle j3 \mapsto 1, j5 \mapsto 0, j? \mapsto j5, j3' \mapsto 1, j5' \mapsto 5\rangle$, $\langle j3 \mapsto 0, j5 \mapsto 1, j? \mapsto j5, j3' \mapsto 0, j5' \mapsto 5\rangle$, and many more. Note that $\langle j3 \mapsto 0, j5 \mapsto 0, j? \mapsto j5, j3' \mapsto 1, j5' \mapsto 5\rangle$ is not a valid replacement, as the same variable x_0 has been replaced by two different values in the same binding. There are 96 resulting operation bindings, which will not be listing here in full. This helps show one important reason for using this type of state diagram, compared to a state diagram that would show every state and transition in full. We have simplified a state diagram that would have nearly 100 transitions to one that has only four. Additionally, this diagram does not visualise the *Transfer_Jar* operation, as the methods we have applied to simplify this

visualisation would not work after this operation is included. Showing all transitions for each operation would require over 1000 transitions. This is why we use handmade, custom, or otherwise simplified visualisations, even though it affects the soundness of the visualisation.

We can now start finding a refinement relation between the specification and the visualisation. Because the state space is the same in the visualisation and specification, we will be checking operation refinement for *Fill_Jar* and *Empty_Jar*.

Calculating the preconditions for the operations gives us two collections of bindings that contain the beforestate and input observation of the operations.

$$\mathbf{pre}\, AFill_Jar = [\, Level;\ j? : Jars \mid level(j?) < max_fill(j?)\,]$$
$$\mathbf{pre}\, AEmpty_Jar = [\, Level;\ j? : Jars \mid level(j?) > 0\,]$$

We can obtain similar collection of bindings for the concrete (visualisation) operations by removing the afterstate observations from the bindings in $\delta Fill_Jar$ and $\delta Empty_Jar$. However, the visualisation shows that it is possible to empty jars that are already empty, and fill jars that are already full, meaning they have weaker preconditions than the abstract operations. So, does the applicability property hold?

$$\forall\, Level;\ j? : Jars \bullet \mathbf{pre}\, AFill_Jar \Rightarrow \mathbf{pre}\, CFill_Jar$$
$$\equiv (Substitution)$$
$$\forall\, Level;\ j? : Jars \bullet \mathbf{pre}\, level(j?) < max_fill(j?) \Rightarrow \mathbf{true}$$
$$\equiv (P \Rightarrow \mathbf{true})$$
$$\mathbf{true}$$

For both operations we can conclude that the applicability property holds as the visualisation operations are enabled in all the beforestates where the abstract operations are enabled and more.

We we be using the standard contractual interpretation when checking correctness. We check that the all behaviour shown in the visualisation is the same as the specification in the states where the behaviour of the operation is specified in the specification. To check this we remove bindings from $\delta Empty_Jar$ where the value of $j?$ in the beforestate is 0. Then, we check that all remaining bindings are in $AEmpty_Jar$. Because the visualisation shows that jar $j?$ is set to 0 while the level of the other jar is unchanged this is true. We can do a similar check for *Fill_Jar* although a more rigorous test would involve testing each binding. This is why in later proofs we use schemas to represent the state diagram.

So, because the visualisation has passed the tests for operation refinement, Figure 6.3 is a sound visualisation of the *Fill_Jar* and *Empty_Jar* operations from the *Jars* example.

Appendix D

Proofs for Figure 6.4

Figure 6.4 is visualisation of the birthday book specification. Once again variables have been used to simplify a large model. With no limits on the size of $NAME$ and $DATE$, this specification has an infinitely large state space. Like the previous example, $(Q, \Sigma, \delta, \delta_0)$ will be written using the invalid bindings directly, followed by a few bindings selected from the infinitely many that are used as the valid replacements.

$$Q = \{\langle bdays \mapsto \{\}\rangle, \langle bdays \mapsto b \cup \{f, newD\}\rangle, \langle bdays \mapsto b \cup \{f, d\}\rangle, \langle bdays \mapsto b\rangle\}$$

The state bindings shown here in Q are not entirely accurate, because this state diagram has an additional requirement that $f \notin \operatorname{dom} b$. So $\langle bdays \mapsto \{(friend1, date1), (friend1, date2)\}\rangle$ is not a valid state binding, although it may appear to be so from how we have written Q above. Remember that this is only a useful intermediary step to work out the real value of Q, which is an infinite set of valid bindings.

$$Q = \{\langle bdays \mapsto \{\}\rangle, \langle bdays \mapsto \{(friend1, date2)\rangle, \langle bdays \mapsto \{(friend1, date3)\rangle, \ldots$$
$$\langle bdays \mapsto \{(friend1, date1)\}\rangle, \langle bdays \mapsto \{(friend1, date1), (friend2, date1)\}\rangle, \ldots$$

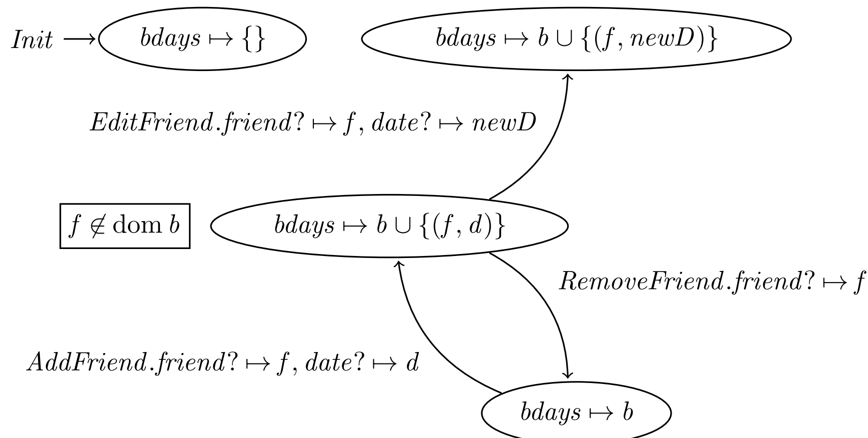


Figure 6.4: State diagram of birthday book (repeated from page 62)

When completely expanded and the observation *known* is included, Q is the same as the state space in the specification. Σ is the set of operations and their input used in the state diagram, this is the same as the specification. $\delta_0 = \langle bdays \mapsto \{\}\rangle$, this matches with the specification as it should.

There are only three transitions in the state diagram, each representing a potentially infinite number of bindings. We look at each of these operations separately, starting with *EditFriend*. $\delta EditFriend = \langle bdays \mapsto b \cup \{(f, d)\}, friend? \mapsto f, date? \mapsto newD, bdays' \mapsto b \cup \{(f, newD)\}\rangle$. When replacing the variables with values, remember to be consistent. For example, when replacing f with $friend1$, all instances of f should be replaced with the same value $friend1$, including in $f \notin \mathrm{dom}\ b$. Like the state bindings, we need to remove the bindings from δ that do not satisfy this additional requirement. This transition only makes one small change to *bdays*, the value of d is changed to $newD$, while the rest is left unchanged. The beforestate of this transition includes all but one, $\langle bdays \mapsto \{\}\rangle$, as $\{\}$ is the only value of *bdays* that cannot satisfy $b \cup \{(f, d)\}$.

$$\delta EditFriend = \langle bdays \mapsto \{(friend1, date1)\}, friend? \mapsto friend1, date? \mapsto date2,$$
$$bdays' \mapsto \{(friend1, date2)\}\rangle,$$
$$\langle bdays \mapsto \{(friend1, date1), (friend2, date1)\}, friend? \mapsto friend1, date? \mapsto date2,$$
$$bdays' \mapsto \{(friend1, date2), (friend2, date1)\}\rangle \ldots$$

Only two of the bindings of $\delta EditFriend$ are shown here to help demonstrate that this transition contains bindings like we would expect.

$\delta RemoveFriend$ removes an element from the *bdays* set, while leaving b unchanged. Thus, this is fairly straightforward to expand.

$$\delta RemoveFriend = \langle bdays \mapsto \{(friend1, date1)\}, friend? \mapsto friend1, bdays' \mapsto \{\}\rangle,$$
$$\langle bdays \mapsto \{(friend1, date1), (friend2, date1)\}, friend? \mapsto friend1,$$
$$bdays' \mapsto b \cup \{(friend2, date1)\}\rangle, \ldots$$

Finally, $\delta AddFriend$ adds inputs (f, d) to *bdays*, without changing any existing birthdays b. There is also the additional requirement that $f \notin \mathrm{dom}\ b$.

$$\delta AddFriend = \langle bdays \mapsto \{\}, friend? \mapsto friend1, date? \mapsto date1, bdays' \mapsto \{(friend1, date1)\}\rangle,$$
$$\langle bdays \mapsto \{(friend2, date1)\}, friend? \mapsto friend1, date? \mapsto date1,$$
$$bdays' \mapsto b \cup \{(friend1, date1)(friend2, date1)\}\rangle, \ldots$$

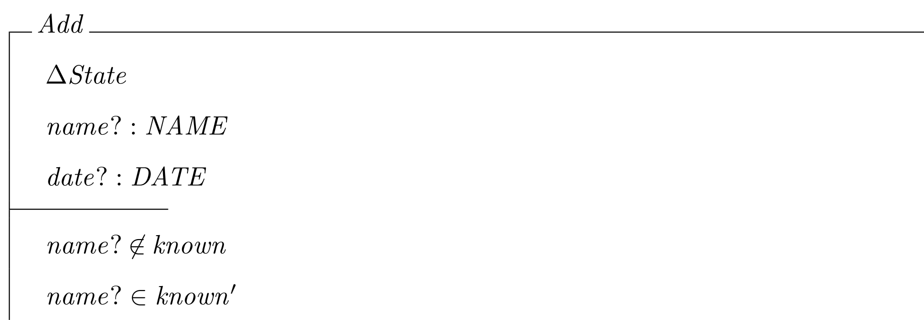
The resulting collections of bindings for δ are exactly the same as the respective operations from the specification, once the observation *known* is included properly. This means that this is a trivial refinement, and the visualisation is sound. However, the method of expanding and checking each binding individually is infeasible. As such, for the following examples we define the state diagrams using schemas rather than bindings.

Appendix E

Proofs for Figure 6.10

Figure 6.10 is a simple visualisation of the birthday book that is used to clearly show the preconditions of the operations. Because the visualisation refers to the specification observations *name*? and *known*, we will begin by assuming that the underlying visualisation state is the same as the specification state. This lets us use operation refinement to determine if the visualisation is sound. If the visualisation referred to observations that were not part of the original specification, then we would use data refinement instead.

We begin with the *Add* operation, which has two input observations, *name*? and *date*?. The visualisation gives us the constraints of this operation, $name? \notin known$ and $name? \in known'$.



Add

$\Delta State$

$name? : NAME$

$date? : DATE$

$name? \notin known$

$name? \in known'$

To use operation refinement, we first find the preconditions of the abstract and concrete

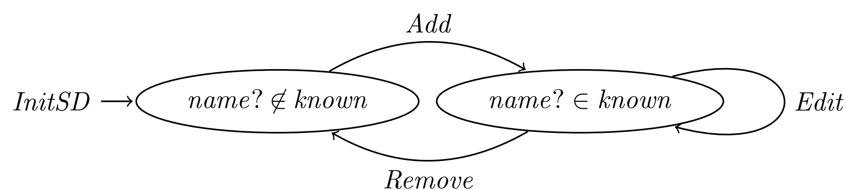


Figure 6.10: Precondition labels on states (repeated from page 66)

operations. Because they are identical, the applicability property is **true** trivially.

$$\textbf{pre}\, Add = [\, State;\ name? : NAME;\ date? : DATE \mid name? \notin known\,]$$

$$\textbf{pre}\, AddFriend = [\, State;\ name? : NAME;\ date? : DATE \mid name? \notin known\,]$$

$$\forall\, State;\ name? : NAME;\ date? : DATE \bullet name? \notin known \Rightarrow name? \notin known$$

$$\equiv (P \Rightarrow P)$$

$$\textbf{true}$$

Then, we need to show the correctness property holds.

$$\forall\, State;\ State';\ ?AOp;\ !AOp \bullet \textbf{pre}\, AOp \wedge COp \Rightarrow AOp$$

$$\equiv (Substitution)$$

$$\forall\, State;\ State';\ name? : NAME;\ date? : DATE \bullet \textbf{pre}\, AddFriend \wedge Add \Rightarrow AddFriend$$

$$\equiv (Substitution)$$

$$\forall\, State;\ State';\ name? : NAME;\ date? : DATE \bullet name? \notin known \wedge name? \in known' \Rightarrow$$
$$name? \notin known \wedge birthdaybook' = birthdaybook \cup name? \mapsto date?$$

Here we notice that the left and right sides of the implication are uneven, so we look for a counterexample. We can also expand *State* and *State'* to reveal that $known = \text{dom}\, birthdaybook$. We only need to find one combination of observation values such that **true** $\Rightarrow$ **false** for the correctness property to be **false**. There are many examples of this, firstly, we start with an empty birthday book where $birthdaybook = \{\}$ and $known = \{\}$. Our input observation values are $name? = A$ and $date? = D$. However, our counterexample states $birthdaybook' = \{(A, D), (B, D)\}$ and $known' = \{A, B\}$. Two names have been added, which is not correct in the specification. Hence, substituting this counterexample into the implication results in **false**.

$$A \notin \{\} \wedge A \in \{A, B\} \Rightarrow A \notin \{\} \wedge \{(A, D), (B, D)\} = \{\} \cup A \mapsto D$$

$$\equiv (\cup rule)$$

$$\textbf{true} \wedge \textbf{true} \Rightarrow \textbf{true} \wedge \textbf{false}$$

$$\equiv (\textbf{true} \Rightarrow \textbf{false})$$

$$\textbf{false}$$

The visualisation is not sound because it does not show that names other than what we are currently adding remain unchanged. In fact, the correctness property also does not hold for *Edit* and *Remove* for the same reason. So, although the visualisation shows the preconditions of the operations accurately, the postconditions are more complex than what is shown in the visualisation.

Appendix F

Proofs for Figure 6.11

The visualisation in Figure 6.11 has merged the specification into three main states, depending on the name Alan Turing. Additionally, the labels on the transitions are also shortened. For example, $Edit_{A,T}$ refers to the edit friend operation being used with $name? = A$ and $date? = T$, while Add_O refers to the add friend operation being used where $name? \neq A$.

We will formalise the operations using schema calculus, where $State0$, $StateT$ and $StateF$ are schemas that refer to the states in the visualisation. This lets us write transitions as follows:

$$Add_O \mathrel{\widehat=} State0 \land State0' \land [\, name? : NAME;\ date? : DATE \mid name? \neq A \,]$$

However, to shorten the proof, rather than include the input values using horizontal schemas we simply show the constraints:

$$Add_O \mathrel{\widehat=} State0 \land State0' \land name? \neq A$$

To use data refinement, we need a retrieve relation that matches the specification states with the visualisation states.

$$\begin{array}{l}
R \\
\hline
State \\
SDState \\
\hline
State0 \land A \notin known \lor \\
StateT \land (A, T) \in birthdaybook \lor \\
StateF \land A \in known \land (A, T) \notin birthdaybook
\end{array}$$

We will begin by formalising Add, then find the precondition and check the applicability

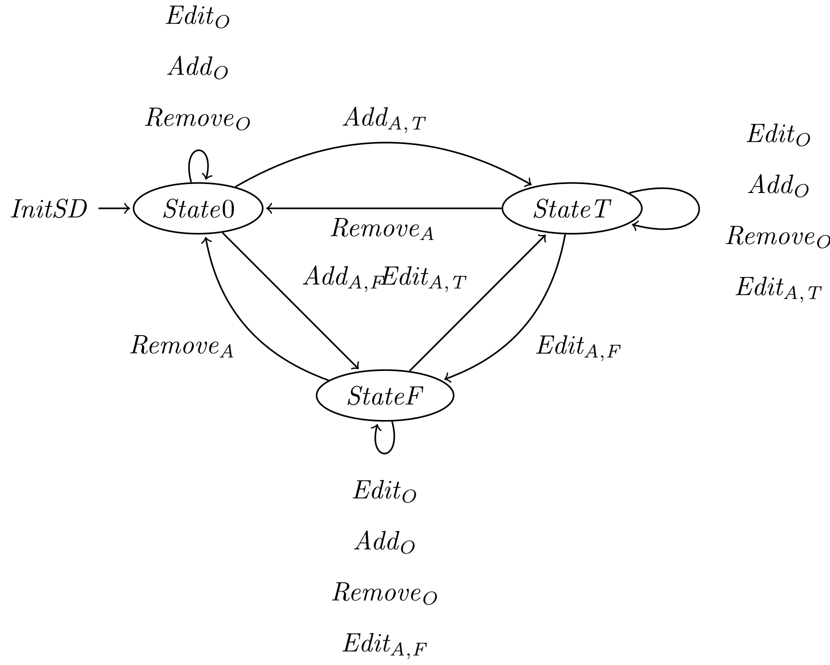


Figure 6.11: Alan Turing Visualisation (repeated from page 67)

and correctness properties.

$$\begin{aligned}
Add \mathrel{\widehat=} & (State0 \land State0' \land name? \neq A) \lor \\
& (StateT \land StateT' \land name? \neq A) \lor \\
& (StateF \land StateF' \land name? \neq A) \lor \\
& (State0 \land StateT' \land name? = A \land date? = T) \lor \\
& (State0 \land StateF' \land name? = A \land date? \neq T)
\end{aligned}$$

$$\mathbf{pre}\, Add = State0 \lor ((StateT \lor StateF) \land name? \neq A)$$

The precondition of Add is the states where A is not in the book or, otherwise, A is not being added. Next, we check for applicability and correctness:

$$\forall AState;\ CState;\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$$

$\equiv$ (Substitution)

$$\forall State;\ SDState;\ name? : NAME;\ date? : DATE \bullet name? \notin known \land R \Rightarrow$$
$$State0 \lor ((StateT \lor StateF) \land name? \neq A)$$

$\equiv$(Substitution, distribution)

$$\forall State;\ SDState;\ name? : NAME;\ date? : DATE \bullet$$
$$(name? \notin known \land State0 \land A \notin known \lor$$
$$name? \notin known \land StateT \land (A, T) \in birthdaybook \lor$$
$$name? \notin known \land StateF \land A \in known \land (A, T) \notin birthdaybook) \Rightarrow$$
$$State0 \lor ((StateT \lor StateF) \land name? \neq A)$$

$\equiv$ (Using $\mathrm{dom}\, birthdaybook = known$ from $State$)

$\forall\, State;\ SDState;\ name? : NAME;\ date? : DATE \bullet$

$(name? \notin known \wedge State0 \wedge A \notin known \vee$

$name? \notin known \wedge StateT \wedge (A, T) \in birthdaybook \wedge name? \neq A \vee$

$name? \notin known \wedge StateF \wedge A \in known \wedge (A, T) \notin birthdaybook) \wedge name? \neq A \Rightarrow$

$State0 \vee ((StateT \vee StateF) \wedge name? \neq A)$

$\equiv (P \wedge Q \Rightarrow P)$

true

$\forall\, AState;\ CState;\ CState';\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \wedge R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i$

$\equiv$ (Substitution)

$\forall\, State;\ SDState;\ SDState';\ name? : NAME;\ date? : DATE;\ \bullet\ name \notin known \wedge$

$(State0 \wedge A \notin known \vee$

$StateT \wedge (A, T) \in birthdaybook \vee$

$StateF \wedge A \in known \wedge (A, T) \notin birthdaybook) \wedge$

$((State0 \wedge State0' \wedge name? \neq A) \vee$

$(StateT \wedge StateT' \wedge name? \neq A) \vee$

$(StateF \wedge StateF' \wedge name? \neq A) \vee$

$(State0 \wedge StateT' \wedge name? = A \wedge date? = T) \vee$

$(State0 \wedge StateF' \wedge name? = A \wedge date? \neq T)) \Rightarrow$

$\exists\, State' \bullet (State0' \wedge A \notin known' \vee$

$StateT' \wedge (A, T) \in birthdaybook' \vee$

$StateF' \wedge A \in known' \wedge (A, T) \notin birthdaybook') \wedge$

$name? \notin known \wedge birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$

$\equiv (DNF, State0 \wedge StateT = \mathbf{false})$

$\forall\, State;\ SDState;\ SDState';\ name? : NAME;\ date? : DATE;\ \bullet\ name? \notin known \wedge$

$((State0 \wedge A \notin known \wedge State0' \wedge name? \neq A) \vee$

$(State0 \wedge A \notin known \wedge StateT' \wedge name? = A \wedge date? = T) \vee$

$(State0 \wedge A \notin known \wedge StateF' \wedge name? = A \wedge date? \neq T) \vee$

$(StateT \wedge (A, T) \in birthdaybook \wedge StateT' \wedge name? \neq A) \vee$

$(StateF \wedge A \in known \wedge (A, T) \notin birthdaybook \wedge StateF' \wedge name? \neq A)) \Rightarrow$

$\exists\, State' \bullet name? \notin known \wedge$

$(State0' \wedge A \notin known' \wedge birthdaybook' = birthdaybook \cup \{name? \mapsto date?\} \vee$

$StateT' \wedge (A, T) \in birthdaybook' \wedge birthdaybook' = birthdaybook \cup \{name? \mapsto date?\} \vee$

$StateF' \wedge A \in known' \wedge (A, T) \notin birthdaybook') \wedge$

$birthdaybook' = birthdaybook \cup name? \mapsto date?$

$\equiv$ (One-point Rule $birthdaybook' = birthdaybook \cup name? \mapsto date?$)

$$\forall State;\ SDState;\ SDState';\ name? : NAME;\ date? : DATE;\ \bullet\ name? \notin known \land$$
$$((State0 \land A \notin known \land State0' \land name? \neq A) \lor$$
$$(State0 \land A \notin known \land StateT' \land name? = A \land date? = T) \lor$$
$$(State0 \land A \notin known \land StateF' \land name? = A \land date? \neq T) \lor$$
$$(StateT \land (A, T) \in birthdaybook \land StateT' \land name? \neq A) \lor$$
$$(StateF \land A \in known \land (A, T) \notin birthdaybook \land StateF' \land name? \neq A)) \Rightarrow$$
$$name? \notin known \land$$
$$(State0' \land A \notin \mathrm{dom}\ birthdaybook \cup \{name? \mapsto date?\} \lor$$
$$StateT' \land (A, T) \in birthdaybook \cup \{name? \mapsto date?\} \lor$$
$$StateF' \land A \in \mathrm{dom}\ birthdaybook \cup \{name? \mapsto date?\} \land$$
$$(A, T) \notin birthdaybook \cup \{name? \mapsto date?\})$$

$\equiv$ $\cup$ Definition, rearrange

$$\forall State;\ SDState;\ SDState';\ name? : NAME;\ date? : DATE;\ \bullet\ name? \notin known \land$$
$$((State0 \land A \notin known \land State0' \land name? \neq A) \lor$$
$$(State0 \land A \notin known \land StateT' \land name? = A \land date? = T) \lor$$
$$(State0 \land A \notin known \land StateF' \land name? = A \land date? \neq T) \lor$$
$$(StateT \land (A, T) \in birthdaybook \land StateT' \land name? \neq A) \lor$$
$$(StateF \land A \in known \land (A, T) \notin birthdaybook \land StateF' \land name? \neq A)) \Rightarrow$$
$$name? \notin known \land$$
$$(A \notin known \land State0' \land name? \neq A \lor$$
$$A \notin known \land StateT' \land name? = A \land date? = T \lor$$
$$A \notin known \land StateF' \land name? \neq A \land (A, T) \notin birthdaybook \land date? \neq T \lor$$
$$(A, T) \in birthdaybook \land StateT' \land name? \neq A \lor$$
$$A \in known \land (A, T) \notin birthdaybook \land StateF' \land name? \neq A)$$

$\equiv$ ($P \land Q \Rightarrow P$)

true

So, the applicability and correctness properties hold for *Add*. Although we do not show the

complete proof, the schema calculus definitions for *Edit* and *Remove* are as follows.

$$
\begin{aligned}
Edit \mathrel{\widehat=}\ & State0 \land State0' \land name? \neq A \lor \\
& StateT \land StateT' \land name? \neq A \lor \\
& StateT \land StateT' \land name? = A \land date? \neq T \lor \\
& StateF \land StateF' \land name? \neq A \lor \\
& StateF \land StateF' \land name? = A \land date? \neq T \lor \\
& StateF \land StateT' \land name? = A \land date? = T \lor \\
& StateT \land StateF' \land name? = A \land date? \neq T \\
Remove \mathrel{\widehat=}\ & State0 \land State0' \land name? \neq A \lor \\
& StateT \land StateT' \land name? \neq A \lor \\
& StateT \land State0' \land name? = A \lor \\
& StateF \land StateF' \land name? \neq A \lor \\
& StateF \land State0' \land name? = A
\end{aligned}
$$

Finally, we check if the initialisation property holds.

$$
\begin{aligned}
& \forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \land R' \\
& \equiv \text{(Substitution)} \\
& \forall\, SDState' \bullet State0' \Rightarrow \exists\, State' \bullet known' = \{\} \land \\
& State0' \land A \notin known' \lor \\
& StateT' \land (A, T) \in birthdaybook' \lor \\
& StateF' \land A \in known' \land (A, T) \notin birthdaybook' \\
& \equiv \text{(One-point rule } known' = \{\}) \\
& \forall\, SDState' \bullet State0' \Rightarrow State0' \\
& \equiv (P \Rightarrow P) \\
& \mathbf{true}
\end{aligned}
$$

Appendix G

Proofs for Figure 6.13

Next, we prove that the visualisation in Figure 6.13 is sound. Because the *Unexplored* state is not a valid state in the original specification we will use data refinement where the retrieve relation merges together the states we are not examining into one state.

R

$State$

$SMState$

$birthdaybook = \{\} \wedge State_{\varnothing} \vee$

$birthdaybook = \{(A, T)\} \wedge State_{AT} \vee$

$birthdaybook = \{(A, F)\} \wedge State_{AF} \vee$

$birthdaybook \neq \{\} \wedge birthdaybook \neq \{(A, T)\} \wedge$

$birthdaybook \neq \{(A, F)\} \wedge Unexplored$

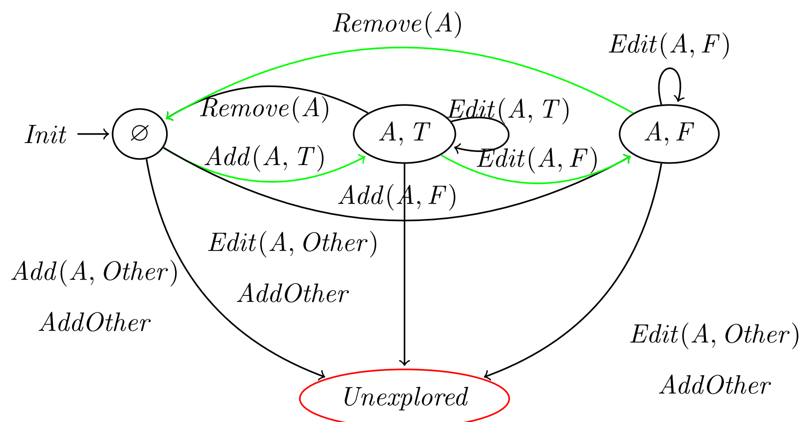


Figure 6.13: Unexplored Alan Turing Visualisation (repeated from page 68)

First we give the preconditions we will be using in the proof:

$$\mathbf{pre}\, Add_Friend_R \equiv birthdaybook = \{\} \lor (birthdaybook = \{(A, T)\} \lor$$
$$birthdaybook = \{(A, F)\}) \land name? \notin known$$
$$\mathbf{pre}\, Remove_Friend_R \equiv (birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\})$$
$$\land\ name? \in known$$
$$\mathbf{pre}\, Edit_Friend_R \equiv (birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\})$$
$$\land\ name? \in known$$
$$\mathbf{pre}\, Add \equiv State_{\varnothing} \lor (State_{AT} \lor State_{AF}) \land name? \neq A$$
$$\mathbf{pre}\, Edit \equiv (State_{AT} \lor State_{AF}) \land name? = A$$
$$\mathbf{pre}\, Remove \equiv (State_{AT} \lor State_{AF}) \land name? = A$$

We can now begin checking refinement. Firstly, checking *Init*.

$$\forall\, SDState' \bullet InitSM \Rightarrow \exists\, State' \bullet Init \land R'$$

$\equiv$(Substitutions)

$$\forall\, SDState' \bullet State'_{\varnothing} \Rightarrow \exists\, birthdaybook' : NAME \nrightarrow DATE, known' : \mathbb{P}\, NAME \mid$$
$$known' = \operatorname{dom} birthdaybook' \bullet known' = \varnothing \land R'$$

$\equiv$(One-point rule, then $State'_{\varnothing}$ since $known$ is empty)

$$\forall\, SDState' : \mathbb{P}\, NAME \bullet State'_{\varnothing} \Rightarrow State'_{\varnothing}$$

$\equiv (P \Rightarrow P)$

true

Since *SDState* says that we can only be in a single state and R gives the requirements on how to be in each state, if those requirements are met we simply replace R with the appropriate state instead of expanding R. Since the initial test passed, we can prove the applicability property for each of the operations.

Firstly, *Add*:

$$\forall\, State;\ SDState;\ name?;\ date? \bullet \mathbf{pre}\, AddFriend_R \land R \Rightarrow \mathbf{pre}\, Add$$

$\equiv$(Substitutions)

$$\forall\, State;\ SDState;\ name?;\ date? \bullet birthdaybook = \{\} \land R \lor$$
$$(birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\}) \land name? \notin known \land R \Rightarrow$$
$$State_{\varnothing} \lor (State_{AT} \lor State_{AF}) \land name? \neq A$$

$\equiv$(Distribute over R and remove **false** cases)

$$\forall\, State;\ SDState;\ name?;\ date? \bullet birthdaybook = \{\} \land State_{\varnothing} \lor$$
$$(birthdaybook = \{(A, T)\} \land State_{AT} \lor birthdaybook = \{(A, F)\} \land State_{AF}) \land$$
$$name? \notin known \Rightarrow State_{\varnothing} \lor (State_{AT} \lor State_{AF}) \land name? \neq A$$

$\equiv$ (Using $known = \mathrm{dom}\, birthdaybook$)

$$\forall State;\ SDState;\ name?;\ date? \bullet birthdaybook = \{\} \land State_{\varnothing} \lor$$
$$(birthdaybook = \{(A, T)\} \land State_{AT} \lor birthdaybook = \{(A, F)\} \land State_{AF}) \land$$
$$name? \notin known \land name? \neq A \Rightarrow$$
$$State_{\varnothing} \lor (State_{AT} \lor State_{AF}) \land name? \neq A$$

$\equiv$ ($P \land Q \Rightarrow P$)

true

Secondly, *Remove*:

$$\forall State;\ SDState;\ name?;\ date? \bullet \textbf{pre}\, RemoveFriend_R \land R \Rightarrow \textbf{pre}\, Remove$$

$\equiv$(Substitutions)

$$\forall birthdaybook;\ known;\ SDState;\ name?;\ date? \bullet (birthdaybook = \{(A, T)\} \lor$$
$$birthdaybook = \{(A, F)\}) \land name? \in known \land R \Rightarrow$$
$$(State_{AT} \lor State_{AF}) \land name? = A$$

$\equiv$(Distribute over R and remove **false** cases)

$$\forall birthdaybook;\ known;\ SDState;\ name?;\ date? \bullet (birthdaybook = \{(A, T)\} \land$$
$$State_{AT} \lor birthdaybook = \{(A, F)\} \land State_{AF}) \land name? \in known \Rightarrow$$
$$(State_{AT} \lor State_{AF}) \land name? = A$$

$\equiv$ (Using $known = \mathrm{dom}\, birthdaybook$)

$$\forall birthdaybook;\ known;\ SDState;\ name?;\ date? \bullet (birthdaybook = \{(A, T)\} \land$$
$$State_{AT} \lor birthdaybook = \{(A, F)\} \land State_{AF}) \land name? \in known \land name? = A$$
$$\Rightarrow (State_{AT} \lor State_{AF}) \land name? = A$$

$\equiv$ ($P \land Q \Rightarrow P$)

true

Finally, *Edit* has the same proof as *Remove*:

$$\forall State;\ SDState;\ name?;\ date? \bullet \textbf{pre}\, EditFriend_R \land R \Rightarrow \textbf{pre}\, Edit$$

$\equiv$(Substitutions)

$$\forall birthdaybook;\ known;\ SDState;\ name?;\ date? \bullet (birthdaybook = \{(A, T)\} \lor$$
$$birthdaybook = \{(A, F)\}) \land name? \in known \land R \Rightarrow$$
$$(State_{AT} \lor State_{AF}) \land name? = A$$

$\equiv$(Distribute over R and remove **false** cases)

$\forall \mathit{birthdaybook}; \ \mathit{known}; \ \mathit{SDState}; \ \mathit{name?}; \ \mathit{date?} \bullet (\mathit{birthdaybook} = \{(A, T)\} \land$
$\mathit{State}_{AT} \lor \mathit{birthdaybook} = \{(A, F)\} \land \mathit{State}_{AF}) \land \mathit{name?} \in \mathit{known} \Rightarrow$
$(\mathit{State}_{AT} \lor \mathit{State}_{AF}) \land \mathit{name?} = A$

$\equiv$ (Using $\mathit{known} = \mathrm{dom}\, \mathit{birthdaybook}$)

$\forall \mathit{birthdaybook}; \ \mathit{known}; \ \mathit{SDState}; \ \mathit{name?}; \ \mathit{date?} \bullet (\mathit{birthdaybook} = \{(A, T)\} \land$
$\mathit{State}_{AT} \lor \mathit{birthdaybook} = \{(A, F)\} \land \mathit{State}_{AF}) \land \mathit{name?} \in \mathit{known} \land \mathit{name?} = A \Rightarrow$
$(\mathit{State}_{AT} \lor \mathit{State}_{AF}) \land \mathit{name?} = A$

$\equiv (P \land Q \Rightarrow P)$

true

Next we will investigate contractual correctness for the visualisation.

$\forall \mathit{State}; \ \mathit{SDState}; \ \mathit{SDState}'; \ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow \exists \mathit{State}' \bullet R' \land AOp_i$

Firstly, *Add*:

$\forall \mathit{State}; \ \mathit{SDState}; \ \mathit{SDState}'; \ \mathit{name?}; \ \mathit{date?} \bullet (\mathit{birthdaybook} = \{\} \lor$
$(\mathit{birthdaybook} = \{(A, T)\} \lor \mathit{birthdaybook} = \{(A, F)\}) \land \mathit{name?} \notin \mathit{known}) \land R \land$
$(\mathit{State}_{\varnothing} \land \mathit{State}'_{AT} \land \mathit{name?} = A \land \mathit{date?} = T \lor \mathit{State}_{\varnothing} \land \mathit{name?} = A \land \mathit{date?} = F \land$
$\mathit{State}'_{AF} \lor \mathit{State}_{\varnothing} \land \mathit{name?} \neq A \land \mathit{Unexplored}' \lor$
$\mathit{State}_{AT} \land \mathit{name?} \neq A \land \mathit{Unexplored}' \lor \mathit{State}_{AF} \land \mathit{name?} \neq A \land \mathit{Unexplored}') \Rightarrow$
$\exists \mathit{birthdaybook}'; \ \mathit{known}' \mid \mathit{known}' = \mathrm{dom}\, \mathit{birthdaybook}' \bullet R' \land (\mathit{birthdaybook} = \{\} \lor$
$\mathit{birthdaybook} = \{(A, T)\} \lor \mathit{birthdaybook} = \{(A, F)\}) \land \mathit{name?} \notin \mathit{known} \land$
$\mathit{birthdaybook}' = \mathit{birthdaybook} \cup \{\mathit{name?} \mapsto \mathit{date?}\}$

$\equiv$ {One-point rule, Distribute over R and remove **false** cases}

$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name?;\ date? \bullet (birthdaybook = \{\} \wedge$
$State_{\varnothing} \vee birthdaybook = \{(A, T)\} \wedge State_{AT} \wedge name? \notin known \vee$
$birthdaybook = \{(A, F)\} \wedge State_{AF} \wedge name? \notin known) \wedge$
$(State_{\varnothing} \wedge birthdaybook = \{\} \wedge State'_{AT} \wedge name? = A \wedge date? = T \vee$
$State_{\varnothing} \wedge birthdaybook = \{\} \wedge name? = A \wedge date? = F \wedge State'_{AF} \vee$
$State_{\varnothing} \wedge birthdaybook = \{\} \wedge name? \neq A \wedge Unexplored' \vee$
$State_{AT} \wedge birthdaybook = \{(A, T)\} \wedge name? \neq A \wedge Unexplored' \vee$
$State_{AF} \wedge birthdaybook = \{(A, F)\} \wedge name? \neq A \wedge Unexplored') \Rightarrow$
$(birthdaybook = \{\} \vee birthdaybook = \{(A, T)\} \wedge name? \notin known \vee$
$birthdaybook = \{(A, F)\} \wedge name? \notin known) \wedge$
$(birthdaybook \cup \{name? \mapsto date?\} = \{(A, T)\} \wedge State'_{AT} \vee$
$birthdaybook \cup \{name? \mapsto date?\} = \{(A, F)\} \wedge State'_{AF} \vee$
$birthdaybook \cup \{name? \mapsto date?\} \neq \varnothing \wedge birthdaybook \cup \{name? \mapsto date?\} \neq \{(A, T)\} \wedge$
$birthdaybook \cup \{name? \mapsto date?\} \neq \{(A, F)\} \wedge Unexplored')$

$\equiv (\cup)$

$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name?;\ date? \bullet (birthdaybook = \{\} \wedge$
$State_{\varnothing} \vee birthdaybook = \{(A, T)\} \wedge State_{AT} \wedge name? \notin known \vee$
$birthdaybook = \{(A, F)\} \wedge State_{AF} \wedge name? \notin known) \wedge$
$(birthdaybook \cup \{name? \mapsto date?\} = \{(A, T)\} \wedge State_{\varnothing} \wedge birthdaybook = \{\} \wedge$
$State'_{AT} \wedge name? = A \wedge date? = T \vee$
$birthdaybook \cup \{name? \mapsto date?\} = \{(A, F)\} \wedge State_{\varnothing} \wedge birthdaybook = \{\} \wedge$
$name? = A \wedge date? = F \wedge State'_{AF} \vee$
$(birthdaybook \cup \{name? \mapsto date?\} \neq \{(A, T)\} \wedge$
$birthdaybook \cup \{name? \mapsto date?\} \neq \{(A, F)\} \wedge$
$(State_{\varnothing} \wedge birthdaybook = \{\} \wedge name? \neq A \wedge Unexplored' \vee State_{AT} \wedge$
$birthdaybook = \{(A, T)\} \wedge name? \neq A \wedge Unexplored' \vee$
$State_{AF} \wedge birthdaybook = \{(A, F)\} \wedge name? \neq A \wedge Unexplored'))) \Rightarrow$
$(birthdaybook = \{\} \vee birthdaybook = \{(A, T)\} \wedge name? \notin known \vee$
$birthdaybook = \{(A, F)\} \wedge name? \notin known) \wedge$
$(birthdaybook \cup \{name? \mapsto date?\} = \{(A, T)\} \wedge State'_{AT} \vee$
$birthdaybook \cup \{name? \mapsto date?\} = \{(A, F)\} \wedge State'_{AF} \vee$
$birthdaybook \cup \{name? \mapsto date?\} \neq \{(A, T)\} \wedge$
$birthdaybook \cup \{name? \mapsto date?\} \neq \{(A, F)\} \wedge Unexplored')$

$\equiv (P \wedge Q \Rightarrow P)$

true

Every possible transition in Add_Friend_R, the right hand side of the implication, is shown in the visualisation.

Next, we check contractual correctness for *Remove*. *Remove* only has two transitions in the visualisation.

$$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name? \bullet (birthdaybook = \{(A, T)\} \lor$$
$$birthdaybook = \{(A, F)\}) \land name? \in known \land R$$
$$name? = A \land (State_{AT} \lor State_{AF}) \land State'_{\varnothing} \Rightarrow$$
$$\exists\, birthdaybook';\ known';\ \mid known' = \mathrm{dom}\, birthdaybook' \bullet R' \land$$
$$(birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\}) \land name? \in known \land$$
$$birthdaybook' = \{name?\} \lhd birthdaybook$$

$\equiv$\{One-point rule\}

$$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name? \bullet (birthdaybook = \{(A, T)\} \land$$
$$State_{AT} \lor birthdaybook = \{(A, F)\} \land State_{AF}) \land name? \in known \land name? = A \land$$
$$(State_{AT} \land birthdaybook = \{(A, T)\} \lor State_{AF} \land birthdaybook = \{(A, F)\}) \land State'_{\varnothing}$$
$$\Rightarrow (\{name?\} \lhd birthdaybook = \{\} \land State'_{\varnothing} \land (birthdaybook = \{(A, T)\} \lor$$
$$birthdaybook = \{(A, F)\}) \land name? \in known$$

$\equiv (P \land Q \Rightarrow P$ move $name? = A$ to right)

$$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name? \bullet (birthdaybook = \{(A, T)\} \land$$
$$State_{AT} \lor birthdaybook = \{(A, F)\} \land State_{AF}) \land name? \in known \land name? = A \land State'_{\varnothing}$$
$$\Rightarrow State'_{\varnothing} \land (birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\})$$

$\equiv (P \land Q \Rightarrow P)$

true

Finally, we can also show that the correctness property holds for *Edit*.

$$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name?;\ date? \bullet$$
$$(birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\}) \land name? \in known \land R \land$$
$$name? = A \land (State_{AT} \lor State_{AF}) \land (date? = T \land State'_{AT} \lor$$
$$date? = F \land State'_{AF} \lor date? \neq T \land date? \neq F \land Unexplored') \Rightarrow$$
$$\exists\, birthdaybook';\ known';\ \mid known' = \mathrm{dom}\, birthdaybook' \bullet R' \land$$
$$(birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\}) \land$$
$$name? \in known \land birthdaybook' = birthdaybook \oplus name? \mapsto date?$$

$\equiv$\{One-point rule\}

$$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name?;\ date? \bullet$$
$$(birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\}) \land name? \in known \land R \land$$
$$name? = A \land (State_{AT} \lor State_{AF}) \land (date? = T \land State'_{AT} \lor$$
$$date? = F \land State'_{AF} \lor date? \neq T \land date? \neq F \land Unexplored') \Rightarrow$$
$$(birthdaybook \oplus name? \mapsto date? = \varnothing \land State'_{\varnothing} \lor$$
$$birthdaybook \oplus name? \mapsto date? = \{(A, T)\} \land State'_{AT} \lor$$
$$birthdaybook \oplus name? \mapsto date? = \{(A, F)\} \land State'_{AF} \lor$$
$$birthdaybook \oplus name? \mapsto date? \neq \{(A, T)\} \land$$
$$birthdaybook \oplus name? \mapsto date? \neq \{(A, F)\}$$
$$\land\, Unexplored') \land (birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\})$$
$$\land\, name? \in known$$

$\equiv$(Simplification and $P \Rightarrow P \land Q$)

$$\forall\, birthdaybook;\ known;\ SDState;\ SDState';\ name?;\ date? \bullet$$
$$(birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\}) \land name? \in known \land$$
$$name? = A \land (State_{AT} \lor State_{AF}) \land$$
$$(date? = T \land State'_{AT} \land birthdaybook \oplus A \mapsto date? = \{(A, T)\} \lor$$
$$date? = F \land State'_{AF} \land birthdaybook \oplus A \mapsto date? = \{(A, F)\} \lor$$
$$date? \neq T \land birthdaybook \oplus A \mapsto date? \neq \{(A, T)\} \land$$
$$date? \neq F \land birthdaybook \oplus A \mapsto date? \neq \{(A, F)\} \land Unexplored') \Rightarrow$$
$$(birthdaybook \oplus A \mapsto date? = \{(A, T)\} \land State'_{AT} \lor$$
$$birthdaybook \oplus A \mapsto date? = \{(A, F)\} \land State'_{AF} \lor$$
$$birthdaybook \oplus A \mapsto date? \neq \{(A, T)\} \land$$
$$birthdaybook \oplus A \mapsto date? \neq \{(A, F)\}$$
$$\land\, Unexplored')$$

$\equiv (P \land Q \Rightarrow P)$

true

We have now checked all the data refinement rules for this partial visualisation. We have found a refinement relation and conclude that this visualisation is sound.

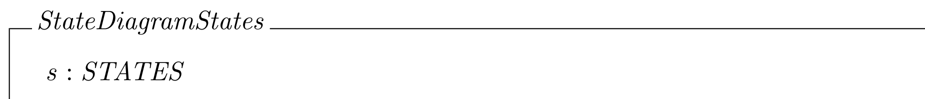
Appendix H

Proofs for Figure 6.15

This is a partial visualisation of the Stopwatch example as it only shows the *Tick* and *Pause/Play* operations. Previously we used schema conjuction to build the transition schemas. We begin this section by showing an alternative approach, using schema inclusion, before checking if the applicability and correctness properties hold for *Tick* and *Pause/Play*.

The given set *STATES* contains the states in the visualisation, *Paused* and *Playing*

$[STATES]$



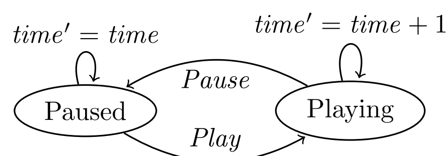


Figure 6.15: These diagram states each represent several specification states (repeated from page 70)

Using *StateDiagramStates* we can then construct the operation schemas.

$$\begin{array}{l}
\textit{Pause/PlayOperation} \\
\hline
\Delta \textit{StateDiagramStates} \\
\hline
\textit{PausedState} \land \textit{PlayingState}' \lor \\
\textit{PlayingState} \land \textit{PausedState}' \\
\hline
\end{array}$$

Note below that *TickOperation* uses a different state space than *Pause/PlayOperation* as it includes the *time* and $time'$ observations.

$$\begin{array}{l}
\textit{TickOperation} \\
\hline
\Delta \textit{StateDiagramStates} \\
\textit{time}, \textit{time}' : \mathbb{N} \\
\hline
\textit{PlayingState} \land \textit{PlayingState}' \land \textit{time}' = \textit{time} + 1 \lor \\
\textit{PausedState} \land \textit{PausedState}' \land \textit{time}' = \textit{time} \\
\hline
\end{array}$$

The precondition of both operations is **true**.

We will use data refinement for this example because the state space of the specification and visualisation are different. We require a retrieve relation schema that relates the two state spaces.

$$\begin{array}{l}
R \\
\hline
\textit{Stopwatch} \\
\textit{StateDiagramStates} \\
\hline
\textit{playing} = \mathbf{false} \land \textit{PausedState} \lor \\
\textit{playing} = \mathbf{true} \land \textit{PlayingState} \\
\hline
\end{array}$$

Because the visualisation does not show initialisation we will begin by checking that applicability holds for each of the operations. Firstly *Pause/Play*: $\forall AState;\ CState;\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$

$$\forall \textit{Stopwatch};\ \textit{StateDiagramStates} \bullet \textit{time} \leq \textit{maxTime} \land R \Rightarrow \mathbf{true}$$
$$\equiv (P \Rightarrow \mathbf{true})$$
$$\mathbf{true}$$

Secondly *Tick*. Note that the precondition of *Tick* is $[\textit{Stopwatch} \mid \textit{playing} = \mathbf{true} \land \textit{time} + 1 \leq \textit{maxTime} \lor \textit{playing} = \mathbf{false} \land \textit{time} \leq \textit{maxTime}]$. Additionally we don't

explicitly quantify over *time* even though it is in the *TickOperation* declarations because it is already in *Stopwatch*.

$$\forall\, Stopwatch;\ StateDiagramStates \bullet playing = \mathbf{true} \land time + 1 \leq maxTime \lor$$
$$playing = \mathbf{false} \land time \leq maxTime \land R \Rightarrow \mathbf{true}$$
$$\equiv (P \Rightarrow \mathbf{true})$$
$$\mathbf{true}$$

Next we will check for correctness, starting with *Pause/Play*.

$$\forall\, State;\ SDState;\ SDState';\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow$$
$$\exists\, State' \bullet R' \land AOp_i$$

$$\forall\, Stopwatch;\ StateDiagramStates;\ StateDiagramStates' \bullet time \leq maxTime \land$$
$$\mathbf{pre}\, Pause/Play \land R \land (PausedState \land PlayingState' \lor PlayingState \land PausedState')$$
$$\Rightarrow \exists\, Stopwatch' \bullet R' \land time' = 0 \land playing' = \neg\, playing$$

$\equiv$(Expand R, R')

$$\forall\, Stopwatch;\ StateDiagramStates;\ StateDiagramStates' \bullet time \leq maxTime \land$$
$$\mathbf{pre}\, Pause/Play \land (playing = \mathbf{false} \land PausedState \land PlayingState' \lor$$
$$playing = \mathbf{true} \land PlayingState \land PausedState') \Rightarrow$$
$$\exists\, Stopwatch' \bullet time' = 0 \land playing' = \neg\, playing \land$$
$$(playing' = \mathbf{false} \land PausedState' \lor playing' = \mathbf{true} \land PlayingState')$$

$\equiv$ (One-point Rule)

$$\forall\, Stopwatch;\ StateDiagramStates;\ StateDiagramStates' \bullet time \leq maxTime \land$$
$$\mathbf{pre}\, Pause/Play \land (playing = \mathbf{false} \land PausedState \land PlayingState' \lor$$
$$playing = \mathbf{true} \land PlayingState \land PausedState') \Rightarrow$$
$$(\neg\, playing = \mathbf{false} \land PausedState' \lor \neg\, playing = \mathbf{true} \land PlayingState')$$

$\equiv$ (Negation)

$$\forall\, Stopwatch;\ StateDiagramStates;\ StateDiagramStates' \bullet time \leq maxTime \land$$
$$\mathbf{pre}\, Pause/Play \land (playing = \mathbf{false} \land PausedState \land PlayingState' \lor$$
$$playing = \mathbf{true} \land PlayingState \land PausedState') \Rightarrow$$
$$(playing = \mathbf{true} \land PausedState' \lor playing = \mathbf{false} \land PlayingState')$$

$\equiv P \land Q \Rightarrow Q$

$$\mathbf{true}$$

Next we will investigate contractual correctness for *Tick*. Unfortunately there is a problem as the $time'$ observation is in both the abstract and concrete state. This means that

the correctness property will be written like $\forall time' : \mathbb{N} \bullet \ldots \exists time' : \mathbb{N} \ldots$, which is invalid. To avoid this problem we rename $time'$ in the concrete state space like $TickOperation[time/_time, time'/_time']$ and include it in the retrieve relation.

$$
\begin{array}{|l}
_R \\
\hline
Stopwatch \\
StateDiagramStates \\
_time : \mathbb{N} \\
\hline
time = _time \\
playing = \mathbf{false} \land PausedState \lor \\
playing = \mathbf{true} \land PlayingState \\
\hline
\end{array}
$$

$$\forall State;\ SDState;\ SDState';\ ?AOp_i \bullet \mathbf{pre}\, AOp_i \land _R \land COp_i \Rightarrow \exists State' \bullet _R' \land AOp_i$$

$$
\begin{array}{l}
\forall Stopwatch;\ StateDiagramStates;\ StateDiagramStates'[time'/_time'] \bullet \\
\quad time \leq maxTime \land \mathbf{pre}\ Tick \land _R \land \\
\quad PlayingState \land PlayingState' \land time' = time + 1 \lor \\
\quad PausedState \land PausedState' \land time' = time \\
\quad \Rightarrow \exists Stopwatch' \bullet _R' \land (playing = \mathbf{true} \land time' = time + 1 \lor \\
\quad playing = \mathbf{false} \land time' = time) \land playing = playing'
\end{array}
$$

$\equiv$(Expand $_R, _R'$, One-point rule)

$$
\begin{array}{l}
\forall Stopwatch;\ StateDiagramStates;\ StateDiagramStates'[time'/_time'] \bullet \\
\quad time \leq maxTime \land \mathbf{pre}\ Tick \land time = _time \land \\
\quad (playing = \mathbf{true} \land PlayingState \land PlayingState' \land _time' = time + 1 \lor \\
\quad playing = \mathbf{false} \land PausedState \land PausedState' \land _time' = time) \\
\quad \Rightarrow \exists time' : \mathbb{N} \bullet time' = _time' \land \\
\quad (PlayingState' \land playing = \mathbf{true} \land time' = time + 1 \lor \\
\quad PausedState' \land playing = \mathbf{false} \land time' = time)
\end{array}
$$

$\equiv$(One-point rule)

$$
\begin{array}{l}
\forall Stopwatch;\ StateDiagramStates;\ StateDiagramStates'[time'/_time'] \bullet \\
\quad time \leq maxTime \land \mathbf{pre}\ Tick \land time = _time \land \\
\quad (playing = \mathbf{true} \land PlayingState \land PlayingState' \land _time' = time + 1 \lor \\
\quad playing = \mathbf{false} \land PausedState \land PausedState' \land _time' = time) \\
\quad \Rightarrow (PlayingState' \land playing = \mathbf{true} \land _time' = time + 1 \lor \\
\quad PausedState' \land playing = \mathbf{false} \land _time' = time)
\end{array}
$$

$\equiv P \land Q \Rightarrow Q$

true

So, the applicabilty and correctness properties hold for these two operations.

H.1 Initialisation Property

We have not checked the initialisation property because the visualisation has no initial state. This leaves our proofs unfinished so we will address this now. There are different ways we can look at this problem because the visualisation presented does not have a fixed formal semantics and so people can interpret it in different ways. We present different options that could be used. Firstly, we define the *CInit* schema as $[StateDiagramStates \mid \textbf{false}]$ This lets us easily prove that the initialisation property holds:

$$\forall\, CState' \bullet \textbf{false} \Rightarrow \exists\, AState' \bullet AInit \land R'$$

$\equiv \textbf{false} \Rightarrow Q$

true

Secondly, if the definition of the visualisation requires an initial state then the visualisation must have an initial state or be invalid. This visualisation does not show an initial state so it would be invalid. However, we can also consider that the visualisation actually has the same initial state as the specification: $[_playing' : BOOL, _time' : \mathbb{N} \mid _playing' = \textbf{false} \land _time' = 0]$. The observation names and R' have been changed to avoid clashes like above.

$$\forall\, _playing' : BOOL;\ _time' : \mathbb{N} \bullet _playing' = \textbf{false} \land _time' = 0 \Rightarrow$$
$$\exists\, playing' : BOOL;\ time' : \mathbb{N} \bullet playing' = \textbf{false} \land time' = 0 \land$$
$$_playing' = playing \land _time' = time$$

$\equiv$(One-Point Rule)

$$\forall\, _playing' : BOOL, _time' : \mathbb{N} \bullet _playing' = \textbf{false} \land _time' = 0 \Rightarrow$$
$$_playing' = \textbf{false} \land _time' = 0$$

$\equiv$

true

While this does work, the state space that is initialised is different from what the visualisation actually operates on. So, if we instead use the previous retreive relation $_R$ and $CInit \mathrel{\widehat=}$

$[StateDiagramStates \mid PausedState']$ schema then we can show the following:

$$\forall\, StateDiagramStates';\ _time' : \mathbb{N} \bullet PausedState' \Rightarrow$$
$$\exists\, playing' : BOOL, time' : \mathbb{N} \bullet (playing' = \mathbf{false} \wedge time' = 0 \wedge$$
$$playing' = \mathbf{false} \wedge PausedState' \vee \mathbf{false}) \wedge _time' = time'$$

$\equiv$(One-Point Rule)

$$\forall\, s' : STATES, _time' : \mathbb{N} \bullet PausedState' \Rightarrow$$
$$PausedState' \wedge _time' = 0$$

This does not hold when the initial state is paused but $_time$, whose value is not explicitly shown in the visualisation, is not 0.

However, if we use the retrieve relation R that does not include *time* then we see the following:

$$\forall\, StateDiagramStates' \bullet PausedState' \Rightarrow$$
$$\exists\, playing' : BOOL, time' : \mathbb{N} \bullet (playing' = \mathbf{false} \wedge time' = 0 \wedge$$
$$playing' = \mathbf{false} \wedge PausedState' \vee \mathbf{false})$$

$\equiv$(One-Point Rule)

$$\forall\, s' : STATES, _time' : \mathbb{N} \bullet PausedState' \Rightarrow PausedState'$$

$\equiv$

$$\mathbf{true}$$

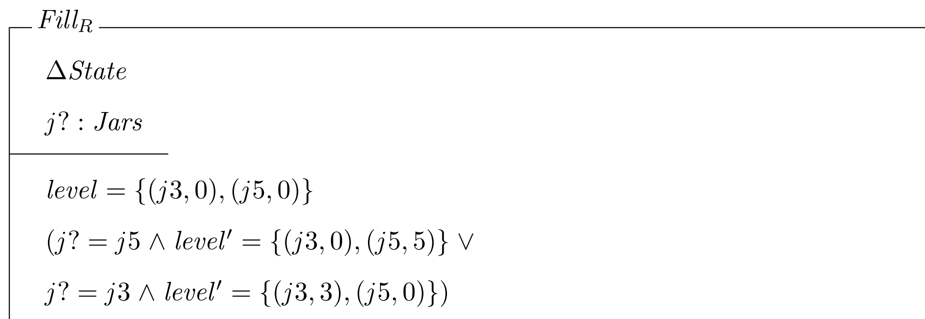
So the initialisation property holds when *Paused* is the initial state for retrieve relation R. This is because the visualisation does not show that the system starts at $_time' = 0$.

Finally, we look at the underlying meaning of the initialisation property to explain why this property holds. Every initial state in the visualisation should match an initial state in the specification. If there is no initial visualisation state or if the set of initial states is the same in both then this property will hold trivially. So, the visualisation will be unsound if the visualisation shows that the system starts in a state that is not possible in the specification. For example, if this visualisation initially began in the *Playing* state or in a state where $_time \neq 0$.

Appendix I

Proofs for Figure 7.3

We use the partial visualisation in Figure 7.3 as an example of how using different restrictions can affect soundness. We use operation refinement for these examples because the visualisation has the same state space as the restricted specification. We begin by considering this as a visualisation of four specification transitions. That is, if any of these transitions are not possible in the specification then the visualisation is unsound. The following schema $Fill_R$ shows the specification operation $Fill_Jar$ after the restriction $level = \{(j3, 0), (j5, 0)\}$ has been added and simplified.



$Fill_R$

$\Delta State$

$j? : Jars$

$level = \{(j3, 0), (j5, 0)\}$

$(j? = j5 \land level' = \{(j3, 0), (j5, 5)\} \lor$

$j? = j3 \land level' = \{(j3, 3), (j5, 0)\})$

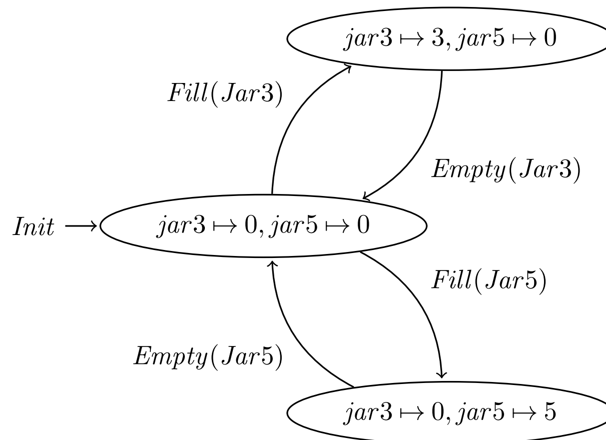


Figure 7.3: Filling and emptying the jar 1 (repeated from page 81)

The precondition of this operation is $[\, State;\ j? : Jars \mid level = \{(j3,0),(j5,0)\}\,]$.

$$\forall\, State;\ j? : Jars \bullet \mathbf{pre}\, Fill_R \Rightarrow \mathbf{pre}\, Fill$$

$\equiv$(Substitutions)

$$\forall\, State;\ j? : Jars \bullet level = \{(j3,0),(j5,0)\} \Rightarrow level = \{(j3,0),(j5,0)\}$$

$\equiv (P \Rightarrow P)$

true

$$\forall\, State;\ State';\ j? : Jars \bullet \mathbf{pre}\, Fill_R \wedge Fill \Rightarrow Fill_R$$

$\equiv$(Substitutions)

$$\begin{aligned}&\forall\, State;\ State';\ j? : Jars \bullet level = \{(j3,0),(j5,0)\} \wedge\\&(j? = j5 \wedge level' = \{(j3,0),(j5,5)\} \vee j? = j3 \wedge level' = \{(j3,3),(j5,0)\}) \Rightarrow\\&level = \{(j3,0),(j5,0)\} \wedge\\&(j? = j5 \wedge level' = \{(j3,0),(j5,5)\} \vee j? = j3 \wedge level' = \{(j3,3),(j5,0)\}\end{aligned}$$

$\equiv (P \Rightarrow P)$

true

The proofs for the correctness and applicability properties are trivial because the transitions shown in the visualisation are exactly what is possible in the specification.

However, this is not a sound partial visualisation of the three states shown. For example, it is possible to $Fill(Jar5)$ in the top state but this is not shown by the visualisation. This can be shown using the following proof. We begin by restricting *level* in *Fill_Jar* to the three states.

$Fill_R$

$\Delta State$

$j? : Jars$

$$\begin{aligned}&level = \{(j3,0),(j5,0)\} \wedge j? = j5 \wedge level' = \{(j3,0),(j5,5)\} \vee\\&level = \{(j3,0),(j5,0)\} \wedge j? = j3 \wedge level' = \{(j3,3),(j5,0)\} \vee\\&level = \{(j3,3),(j5,0)\} \wedge j? = j5 \wedge level' = \{(j3,3),(j5,5)\} \vee\\&level = \{(j3,0),(j5,5)\} \wedge j? = j3 \wedge level' = \{(j3,3),(j5,5)\}\end{aligned}$$

The precondition of this operation is shown in the following schema:

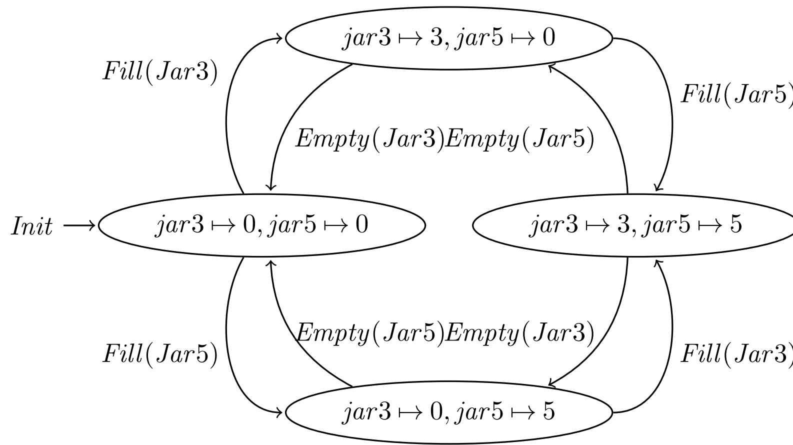


Figure 7.4: Filling and emptying the jar 2 (repeated from page 82)

$Fill_R$

$\Delta State$

$j? : Jars$

$level = \{(j3, 0), (j5, 0)\} \lor$

$level = \{(j3, 3), (j5, 0)\} \land j? = j5 \lor$

$level = \{(j3, 0), (j5, 5)\} \land j? = j3$

$$\forall State;\ j? : Jars \bullet \mathbf{pre}\, Fill_R \Rightarrow \mathbf{pre}\, Fill$$

$\equiv$(Substitutions)

$$\forall State;\ j? : Jars \bullet level = \{(j3, 0), (j5, 0)\} \lor$$
$$level = \{(j3, 3), (j5, 0)\} \land j? = j5 \lor$$
$$level = \{(j3, 0), (j5, 5)\} \land j? = j3 \Rightarrow level = \{(j3, 0), (j5, 0)\}$$

Counterexample ($level = \{(j3, 3), (j5, 0)\} \land j? = j5$)

$$\mathbf{false} \lor \mathbf{true} \land \mathbf{true} \lor \mathbf{false} \land \mathbf{false} \Rightarrow \mathbf{false}$$

$\equiv$ ($\mathbf{true} \Rightarrow \mathbf{false}$)

$$\mathbf{false}$$

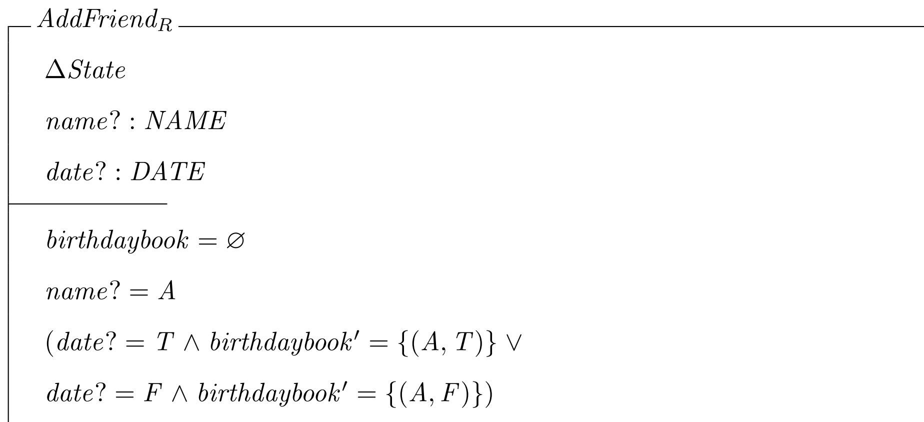
The applicability property does not hold after we weaken the restriction. This is because the visualisation does not show every transition that is possible in the three states. The partial visualisation in Figure 7.4 is a similar example that does show every *Fill* and *Empty* transition that is possible in the four states. Partial visualisations are useful for focusing on a small part of the specification. However, unclear restrictions can be misleading.

Appendix J

Proofs for Figure 7.6

Figure 7.6 shows the same visualisation as Figure 6.13 however the unexplored state has been removed. We will use this example to help show the importance of the unexplored state while also comparing two different restriction strengths. We will focus on the *Add* operation for this example.

We begin by using the strongest reasonable transition restriction. That is, we are restricting the specification to just $Add(A, T)$ and $Add(A, F)$ in the state when $birthdaybook = \varnothing$. This gives us the following schema:



$AddFriend_R$

$\Delta State$

$name? : NAME$

$date? : DATE$

$birthdaybook = \varnothing$

$name? = A$

$(date? = T \wedge birthdaybook' = \{(A, T)\} \vee$

$date? = F \wedge birthdaybook' = \{(A, F)\})$

Next, we show the correctness property holds for these two transitions:

$$\forall\, State;\ State';\ ?AOp_R;\ !AOp_R \bullet \mathbf{pre}\, AOp_R \wedge COp \Rightarrow AOp_R$$

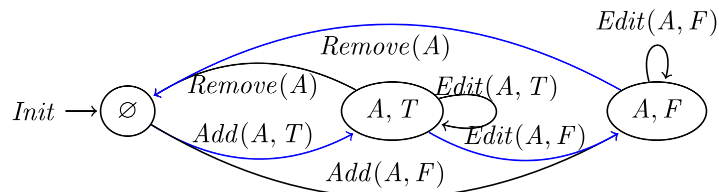


Figure 7.6: Add, Edit, Remove without Unexplored (repeated from page 84)

$\equiv$(Substitutions)

$$\forall\, State;\ State', ?AddFriend_R \bullet (birthdaybook = \varnothing \wedge name? = A \wedge$$
$$(date? = T \vee date? = F)) \wedge (birthdaybook = \varnothing \wedge name? = A \wedge date? = T \wedge$$
$$birthdaybook' = \{(A, T)\} \vee birthdaybook = \varnothing \wedge date? = F \wedge name? = A \wedge$$
$$birthdaybook' = \{(A, F)\}) \Rightarrow$$
$$(birthdaybook = \varnothing \wedge name? = A \wedge date? = T \wedge birthdaybook' = \{(A, T)\} \vee$$
$$birthdaybook = \varnothing \wedge date? = F \wedge name? = A \wedge birthdaybook' = \{(A, F)\})$$

$\equiv (P \wedge Q \Rightarrow P)$

true

Since the correctness property holds we will next check applicability. The preconditions of the operations are identical so this proof is trivial.

$$\forall\, State;\ ?Aop_R \bullet \textbf{pre}\, AOp_R \Rightarrow \textbf{pre}\, COp$$

$\equiv$(Substitutions)

$$\forall\, State;\ ?AddFriend_R \bullet (birthdaybook = \varnothing \wedge name? = A \wedge (date? = T \vee date? = F))$$
$$\Rightarrow (birthdaybook = \varnothing \wedge name? = A \wedge (date? = T \vee date? = F))$$

$\equiv (P \Rightarrow P)$ **true**

Thus, we have shown that *AddFriend* has been visualised soundly when we use the strongest transition restriction. So we know that we can add A to the birthday book when the date is T or F and it will be recorded as shown in the visualisation. However, we would like to use weaker restrictions. Does this visualisation of three states actually visualises the three states or is there missing information?

$AddFriend_R$

$\Delta State$
$name? : NAME$
$date? : DATE$

$(birthdaybook = \varnothing \vee birthdaybook = \{(A, T)\} \vee birthdaybook = \{(A, F)\})$
$name? \notin known$
$birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$

Correctness property:

$$\forall\, State;\ State';\ ?AOp_R;\ !AOp_R \bullet \textbf{pre}\, AOp_R \wedge COp \Rightarrow AOp_R$$

$\equiv$(Substitutions)

$\forall State;\ State';\ ?AddFriend_R \bullet ((birthdaybook = \varnothing \lor birthdaybook = \{(A, T)\} \lor$
$birthdaybook = \{(A, F)\}) \land name? \notin known \land (birthdaybook = \varnothing \land name? = A \land$
$date? = T \land birthdaybook' = \{(A, T)\} \lor birthdaybook = \varnothing \land date? = F \land$
$name? = A \land birthdaybook' = \{(A, F)\}) \Rightarrow$
$(birthdaybook = \varnothing \lor birthdaybook = \{(A, T)\} \lor birthdaybook = \{(A, F)\}) \land$
$name? \notin known \land birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$

$\equiv (P \Rightarrow Q \land P)$

$\forall State;\ State';\ ?AddFriend_R \bullet ((birthdaybook = \varnothing \lor birthdaybook = \{(A, T)\} \lor$
$birthdaybook = \{(A, F)\}) \land name? \notin known \land (birthdaybook = \varnothing \land name? = A \land$
$date? = T \land birthdaybook' = \{(A, T)\} \lor birthdaybook = \varnothing \land date? = F \land$
$name? = A \land birthdaybook' = \{(A, F)\}) \Rightarrow$
$birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$

$\equiv$(Distribution)

$\forall State;\ State';\ ?AddFriend_R \bullet ((birthdaybook = \varnothing \lor birthdaybook = \{(A, T)\} \lor$
$birthdaybook = \{(A, F)\}) \land name? \notin known \land birthdaybook = \varnothing \land name? = A \land$
$((date? = T \land birthdaybook' = \{(A, T)\}) \lor$
$(date? = F \land birthdaybook' = \{(A, F)\})) \Rightarrow$
$birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$

$\equiv (P \Rightarrow Q)$

$\forall State;\ State';\ ?AddFriend_R \bullet ((birthdaybook = \varnothing \lor birthdaybook = \{(A, T)\} \lor$
$birthdaybook = \{(A, F)\}) \land name? \notin known \land birthdaybook = \varnothing \land name? = A \land$
$((date? = T \land birthdaybook' = \{(A, T)\}) \lor$
$(date? = F \land birthdaybook' = \{(A, F)\})) \Rightarrow$
$birthdaybook' = birthdaybook \cup \{name? \mapsto date?\}$
$\land\ birthdaybook = \varnothing \land name? = A \land$
$((date? = T \land birthdaybook' = \{(A, T)\}) \lor$
$(date? = F \land birthdaybook' = \{(A, F)\}))$

$\equiv (\cup)$

$$\forall \mathit{State};\ \mathit{State}';\ ?\mathit{AddFriend}_R \bullet ((\mathit{birthdaybook} = \varnothing \vee \mathit{birthdaybook} = \{(A, T)\} \vee$$
$$\mathit{birthdaybook} = \{(A, F)\}) \wedge \mathit{name}? \notin \mathit{known} \wedge \mathit{birthdaybook} = \varnothing \wedge \mathit{name}? = A \wedge$$
$$((\mathit{date}? = T \wedge \mathit{birthdaybook}' = \{(A, T)\}) \vee$$
$$(\mathit{date}? = F \wedge \mathit{birthdaybook}' = \{(A, F)\})) \Rightarrow$$
$$\mathit{birthdaybook} = \varnothing \wedge \mathit{name}? = A \wedge$$
$$((\mathit{date}? = T \wedge \mathit{birthdaybook}' = \{(A, T)\}) \vee$$
$$(\mathit{date}? = F \wedge \mathit{birthdaybook}' = \{(A, F)\}))$$

$\equiv P \wedge Q \Rightarrow P$

true

The correctness property holds for *AddFriend*. The two transitions in the visualisation have the same behaviour in the specification. However, the visualisation has a much stronger precondition than the specification and this will cause us to be unable to prove the applicability property holds.

$$\forall \mathit{State};\ ?\mathit{Aop}_R \bullet \textbf{pre}\ \mathit{AOp}_R \Rightarrow \textbf{pre}\ \mathit{COp}$$

$\equiv$(Substitutions)

$$\forall \mathit{State};\ ?\mathit{AddFriend}_R \bullet (\mathit{birthdaybook} = \varnothing \vee \mathit{birthdaybook} = \{(A, T)\} \vee$$
$$\mathit{birthdaybook} = \{(A, F)\}) \wedge \mathit{name}? \notin \mathit{known} \Rightarrow$$
$$(\mathit{birthdaybook} = \varnothing \wedge \mathit{name}? = A \wedge (\mathit{date}? = T \vee \mathit{date}? = F))$$

(Counterexample $\mathit{birthdaybook} = \{(A, T)\}, \mathit{name}? = B$)

$$\forall \mathit{date}? : \mathit{DATE}(\{(A, F)\} = \varnothing \vee \{(A, F)\} = \{(A, T)\} \vee$$
$$\{(A, F)\} = \{(A, F)\}) \wedge B \notin \{A\} \Rightarrow$$
$$(\{(A, F)\} = \varnothing \wedge B = A \wedge (\mathit{date}? = T \vee \mathit{date}? = F))$$

$\equiv$(Equality)

$$\forall \mathit{date}? : \mathit{DATE} \bullet (\textbf{false} \vee \textbf{false} \vee \textbf{true}) \wedge \textbf{true} \Rightarrow$$
$$(\textbf{false} \wedge \textbf{false} \wedge (\mathit{date}? = T \vee \mathit{date}? = F))$$

$\equiv$

$$\forall \mathit{date}? : \mathit{DATE} \bullet \textbf{true} \Rightarrow \textbf{false}$$

$\equiv$

false

In our counterexample we used $name? = B$. Because this partial visualisation does not let us add other names to the birthday book from these three states the correctness property does not hold. Because we have removed the *Unexplored* state, we have also removed the transitions entering it, meaning that this is not a partial visualisation of the three states it shows.

Appendix K

Proofs for Figure 7.7

The following is the data refinement proofs for Figure 7.7. We will prove that the conditions hold for each operation in *Jars* when the specification has been restricted to the states we are examining. First, we check *Fill*.

$$\begin{array}{|l}
\multicolumn{1}{l}{Fill_Jar_R} \\
\Delta Level \\
j? : Jars \\
\hline
(level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\}) \\
level(j?) < max_fill(j?) \\
level' = level \oplus \{j? \mapsto max_fill(j?)\} \\
\hline
\end{array}$$

Correctness property:

$$\forall State;\ State';\ ?AOp_R;\ !AOp_R \bullet \mathbf{pre}\, AOp_R \land COp \Rightarrow AOp_R$$

$\equiv$(Substitution)

$$\begin{array}{l}
\forall Level;\ Level', j? : Jars \bullet \\
(level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\}) \land level(j?) < max_fill(j?) \land \\
(level = \{j3 \mapsto 0, j5 \mapsto 0\} \land j? = j3 \land level' = \{j3 \mapsto 3, j5 \mapsto 0\} \lor \\
level = \{j3 \mapsto 0, j5 \mapsto 0\} \land j? = j5 \land level' = \{j3 \mapsto 0, j5 \mapsto 5\} \lor \\
level = \{j3 \mapsto 3, j5 \mapsto 0\} \land j? = j5 \land level' = \{j3 \mapsto 3, j5 \mapsto 5\}) \Rightarrow \\
(level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 0, j5 \mapsto 0\}) \land \\
level(j?) < max_fill(j?) \land level' = level \oplus \{j? \mapsto max_fill(j?)\}
\end{array}$$

$\equiv$

true

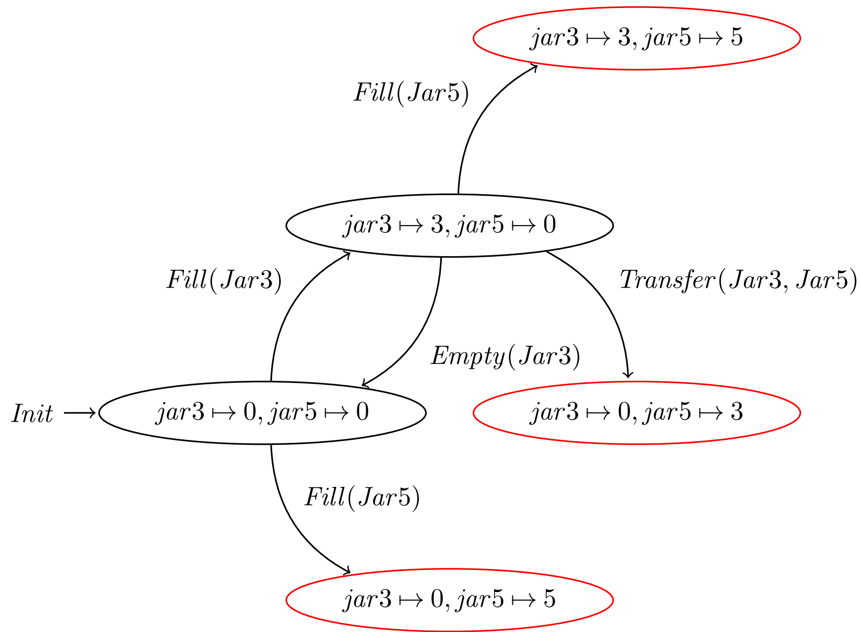


Figure 7.7: Filling and emptying the jar 3 (repeated from page 85)

The three transitions that fill the jar in the visualisation correctly fill the jar as specified by the Z operation.

Applicability property:

$$\forall\, State;\ ?Aop_R \bullet \mathbf{pre}\ AOp_R \Rightarrow \mathbf{pre}\ COp$$

≡(Substitution)

$$\forall\, Level;\ j? : Jars \bullet (level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\}) \land$$
$$level(j?) < max_fill(j?) \Rightarrow$$
$$(level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\} \land j? = j5)$$

≡

true

The applicability property being satisfied means that we are not missing any *Fill_Jar* transitions that should be outgoing from the states we are visualising. Next, we look at *Empty_Jar*, and see that the conditions are satisfied for the same reasons.

$Empty_Jar_R$

$\Delta Level$

$j? : Jars$

$(level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\})$

$level(j?) > 0$

$level' = level \oplus \{j? \mapsto 0\}$

Correctness property:

$$\forall\, State;\ State';\ ?AOp_R;\ !AOp_R \bullet \mathbf{pre}\, AOp_R \wedge COp \Rightarrow AOp_R$$

$\equiv$(Substitution)

$$\forall\, Level;\ Level', j? : Jars \bullet (level = \{j3 \mapsto 0, j5 \mapsto 0\} \vee level = \{j3 \mapsto 3, j5 \mapsto 0\}) \wedge$$
$$level(j?) > 0 \wedge level = \{j3 \mapsto 3, j5 \mapsto 0\} \wedge level' = \{j3 \mapsto 0, j5 \mapsto 0\} \Rightarrow$$
$$(level = \{j3 \mapsto 0, j5 \mapsto 0\} \vee level = \{j3 \mapsto 3, j5 \mapsto 0\}) \wedge$$
$$level(j?) > 0 \wedge level' = level \oplus \{j? \mapsto 0\}$$

$\equiv$

true

Applicability property:

$$\forall\, State;\ ?Aop_R \bullet \mathbf{pre}\, AOp_R \Rightarrow \mathbf{pre}\, COp$$

$\equiv$(Substitution)

$$\forall\, Level;\ j? : Jars \bullet (level = \{j3 \mapsto 0, j5 \mapsto 0\} \vee level = \{j3 \mapsto 3, j5 \mapsto 0\})$$
$$\wedge\ level(j?) > 0 \Rightarrow level = \{j3 \mapsto 3, j5 \mapsto 0\} \wedge j? = j3$$

$\equiv$

true

In the visualisation, there is only one transition that shows the *Empty_Jar* operation being used. As both conditions are satisfied, this means that we have visualised the jar being emptied correctly, and the jars are not emptied in any other way from the two states we are examining.

Finally, we check the conditions hold for *Transfer*.

$Transfer_R$

$\Delta Level$
$j1?, j2? : Jars$
$amount? : \mathbb{N}_1$

$(level = \{j3 \mapsto 0, j5 \mapsto 0\} \vee level = \{j3 \mapsto 3, j5 \mapsto 0\})$
$j1? \neq j2?$
$amount? = min(\{level\, j1?, max_fill\, j2? - level\, j2?\})$
$level' = level \oplus \{j1? \mapsto level\, j1? - amount?, j2? \mapsto level\, j2? + amount?\}$

Correctness property:

$$\forall\, State;\ State';\ ?AOp_R;\ !AOp_R \bullet \mathbf{pre}\, AOp_R \wedge COp \Rightarrow AOp_R$$

≡(Substitution)

$$\forall Level;\ Level', j1?, j2? amount? : \mathbb{N}_1 : Jars \bullet$$
$$(level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\}) \land j1? \neq j2? \land$$
$$amount? = min(\{level\, j1?, max_fill\, j2? - level\, j2?\}) \land$$
$$level = \{j3 \mapsto 3, j5 \mapsto 0\} \land level' = \{j3 \mapsto 0, j5 \mapsto 3\} \land j1? = j3 \land j2? = j5 \Rightarrow$$
$$(level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\}) \land j1? \neq j2? \land$$
$$amount? = min(\{level\, j1?, max_fill\, j2? - level\, j2?\}) \land$$
$$level' = level \oplus \{j1? \mapsto level\, j1? - amount?, j2? \mapsto level\, j2? + amount?\}$$

≡

true

Applicability property:

$$\forall State;\ ?Aop_R \bullet \mathbf{pre}\, AOp_R \Rightarrow \mathbf{pre}\, COp$$

≡(Substitution)

$$\forall Level;\ j? : Jars \bullet (level = \{j3 \mapsto 0, j5 \mapsto 0\} \lor level = \{j3 \mapsto 3, j5 \mapsto 0\}) \land$$
$$j1? \neq j2? \land amount? = min(\{level\, j1?, max_fill\, j2? - level\, j2?\}) \Rightarrow$$
$$level = \{j3 \mapsto 3, j5 \mapsto 0\} \land j1? = j3 \land j2? = j5$$

≡

true

We have shown that the conditions hold for each of the operations in the specification, and so we conclude that this is a sound partial visualisation where the two black states have been fully explored.

Appendix L

Proofs for Figure 8.2

In section 8.6 we presented *SWSys*, the top level operation schema for the μ-chart in Figure 8.2. Here we provide the low-level schemas and axiomatic definitions that were used to construct this schema. Following this we will prove that this visualisation is sound. In section 8.8 we began this with the initialisation property and the *Reset* operation. Here we complete the proofs with the *Pause*/*Play* and *Tick* operations.

There are many low-level schemas. Starting from the lowest level of the chart, we require the following:

1. Schemas for the state space of sequential charts;
2. Seperate state schemas for every state;
3. Init schemas;
4. Operation schemas for each transition in the chart;
5. Inactive transition schemas for each sequential chart;
6. Do-nothing schemas for each sequential chart;
7. Combined transition schemas for each sequential chart;
8. Init, state and transition schemas for the composed chart;
9. Init, state and transition schemas for the decomposed chart;
10. Init, state and transition schemas for the hiding operator;
11. Transition schema for the step semantics.

This, along with appropriate axiomatic definitions is the list of schemas defined in previous works to create the full model of a μ-chart.

When we use this in our example we will not need to create the schemas for the decomposed

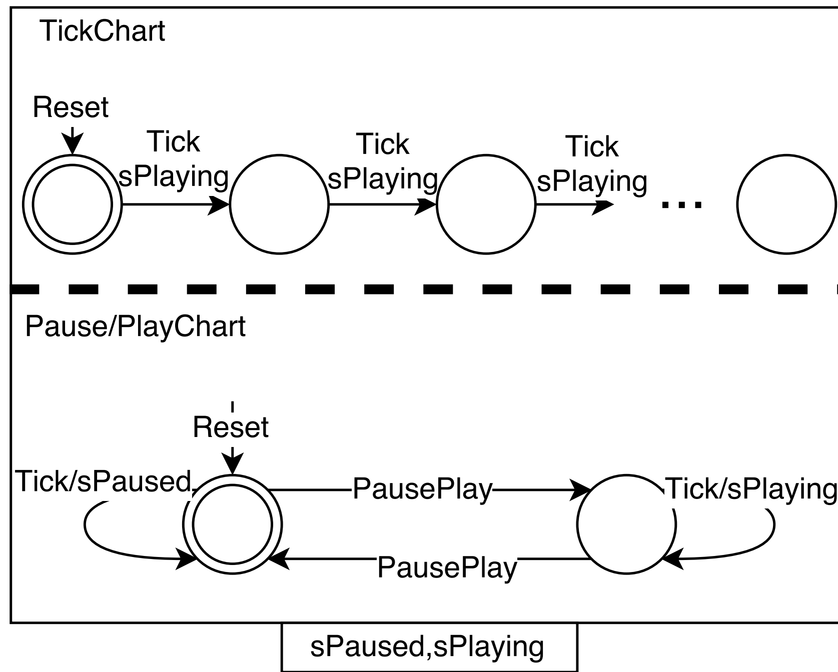


Figure 8.2: Composed Microchart (repeated from page 94)

chart as none are present in our example.

However, we do use the new operation operator to create additional schemas for each operation schema in the original specification, namely *Pause/Play*, *Reset* and *Tick*.

L.1 State, Init and Axiomatic Definitions for Simple Sequential Charts

$$\mu_{States} ::= t \langle\langle \mathbb{N} \rangle\rangle \mid p \langle\langle Bool \rangle\rangle$$
$$\mu_{Signal} ::= op \langle\langle OperationNames \rangle\rangle \mid sPlaying \mid sPaused$$

Here we are defining the states, input, output and feedback of the simple charts.

Note that we have redefined the type of the states and the signals.

The state type is now a boolean value or a number, while the signals in the chart will either be the name of the operation being used, Playing or Paused, which are the signals we are feeding back into the μ-chart to indicate the watch is paused or not.

$$
\begin{array}{|l}
states_{Tick} : \mathbb{P}\, \mu_{States} \\
in_{Tick} : \mathbb{P}\, \mu_{Signal} \\
out_{Tick} : \mathbb{P}\, \mu_{Signal} \\
\Psi : \mathbb{P}\, \mu_{Signal} \\
\hline
states_{Tick} = \sum \\
in_{Tick} = in\ Tick \\
out_{Tick} = out\ Tick
\end{array}
$$

This is the state schema for the Tick chart and if we had any local variables they would also be included in this chart.

$$
\begin{array}{|l}
\hline
Chart_{Tick} \\
c_{Tick} : states_{Tick} \\
\hline
\end{array}
$$

For each state in the μ-chart, we have a schema such that the system is in that state.

$$
\begin{array}{|l}
\hline
Tick_{\sigma} \\
Chart_{Tick} \\
\hline
c_T ick = t\, \sigma \\
\hline
\end{array}
$$

An example of this is $Tick_0$ which specifies the state when time is 0. Note that because we are using a redefined type for the states, we need to cast the value for it to be type $states_{Tick}$. We use t for states in Tick and p for states in Pause/Play.

$$
\begin{array}{|l}
\hline
Tick_0 \\
Chart_{Tick} \\
\hline
c_{Tick} = t\, 0 \\
\hline
\end{array}
$$

The initial state of the sequential μ-chart TickChart is $t\, 0$.

$$
\begin{array}{|l}
\hline
Init_{Tick} \\
Chart_{Tick} \\
\hline
c_{Tick} = t\, 0 \\
\hline
\end{array}
$$

Similarly, we can construct the schemas of the Pause/Play chart.

$states_{PP} : \mathbb{P}\, \mu_{States}$

$in_{PP} : \mathbb{P}\, \mu_{Signal}$

$out_{PP} : \mathbb{P}\, \mu_{Signal}$

$\Psi : \mathbb{P}\, \mu_{Signal}$

$states_{PP} = \sum$

$in_{PP} = in\ PP$

$out_{PP} = out\ PP$

$Chart_{PP}$

$c_{PP} : states_{PP}$

PP_{Paused}

$Chart_{PP}$

$c_{PP} = p\,\mathbf{false}$

$PP_{Playing}$

$Chart_{PP}$

$c_{PP} = p\,\mathbf{true}$

$Init_{PP}$

$Chart_{PP}$

$c_{PP} = p\,\mathbf{false}$

L.2 Operation Schemas for Tick

We will be using the semantic sugar "..." to simplify the operation schemas here, but first we give an example of a single transition. This is the transition from state 0 to state 1.

$\delta_0 1$

$$Tick_0$$
$$Tick_1'$$
$$i_{Tick}? : \mathbb{P}\, in_{Tick}$$
$$active_ : \mathbb{P}\, \mu_{State}$$
$$o_{Tick}! : \mathbb{P}\, out_{Tick}$$

$$active(Tick)$$
$$op\ T \in i_{Tick}?$$
$$Playing \in i_{Tick}? \cup (o_{Tick}! \cap \{sPlaying, sPaused\})$$
$$o_{Tick}! = \{\}$$

There are a large number of schemas similar to the one above where the only difference is the start and end states. Additionally, the after state will always be one greater than the start state. We use this information to combine multiple transitions into the following schema.

δ_{inc}

$$Chart_{Tick}$$
$$i_{Tick}? : \mathbb{P}\, in_{Tick}$$
$$active_ : \mathbb{P}\, \mu_{State}$$
$$o_{Tick}! : \mathbb{P}\, out_{Tick}$$

$$active(Tick)$$
$$op\ T \in i_{Tick}?$$
$$sPlaying \in i_{Tick}? \cup (o_{Tick}! \cap \{sPlaying, sPaused\})$$
$$o_{Tick}! = \{\}$$
$$(t^{\sim}\, c_{Tick}) < maxTime$$
$$c'_{Tick} = t\,(t^{\sim}\, c_{Tick} + 1)$$

Similarly, the transition labelled Reset is also a simplification of a large number of transitions. The large number of transitions combine into δ_{reset}, a transition schema with an afterstate but no defined before state.

$$\delta_{reset}$$

$$
\begin{array}{l}
Tick'_0 \\
i_{Tick}? : \mathbb{P}\, in_{Tick} \\
active_ : \mathbb{P}\, \mu_{State} \\
o_{Tick}! : \mathbb{P}\, out_{Tick} \\
\hline
active(Tick) \\
op\ Reset \in i_{Tick}? \\
o_{Tick}! = \{\}
\end{array}
$$

Although this chart will never be inactive, we also present $Inactive_{Tick}$. If the chart was inactive then it would not change states or output any signals.

$$Inactive_{Tick}$$

$$
\begin{array}{l}
\Xi Chart_{Tick} i_{Tick}? : \mathbb{P}\, in_{Tick} \\
active_ : \mathbb{P}\, \mu_{State} \\
o_{Tick}! : \mathbb{P}\, out_{Tick} \\
\hline
\neg\ active(Tick) \\
o_{Tick}! = \{\}
\end{array}
$$

Because the top chart does not respond to the Pause/Play signal it causes the entire chart to behave chaotically when this signal is recieved. We assume that it should do nothing when such signals received and so we include the ϵ 'do-nothing' schema which causes the μ-chart to behave as expected.

$$\epsilon_{Tick}$$

$$
\begin{array}{l}
\Delta Chart_{Tick} \\
i_{Tick}?, o_{Tick}! : \mathbb{P}\, \mu_{Signal} \\
active_ : \mathbb{P}\, \mu_{State} \\
\hline
active(Tick) \\
c'_{Tick} = c_{Tick} \\
o_{Tick}! = \{\} \\
\neg\ (S_{f1} \wedge \rho(guard_1)) \\
\neg\ (S_{f2} \wedge \rho(guard_2)) \\
\qquad \vdots
\end{array}
$$

We can now create the schema containing all the transitions for this chart, which is simply the disjunction of the previous schemas. $\delta_{Tick} == \delta_{reset} \vee \delta_{inc} \vee Inactive_{Tick} \vee \epsilon_{Tick}$ We

write this schema out in full below. Note the text that ϵ_{Tick} contributes is the Pause/Play operation being used and the Tick operation being used at $maxTime$.

δ_{Tick}

$Chart_{Tick}$

$i_{Tick}? : \mathbb{P}\, in_{Tick}$

$active_ : \mathbb{P}\, \mu_{State}$

$o_{Tick}! : \mathbb{P}\, out_{Tick}$

$($

$active(Tick) \wedge$

$op\ Tick \in i_{Tick}? \wedge$

$sPlaying \in i_{Tick}? \cup (o_{Tick}! \cap \{sPlaying, sPaused\}) \wedge$

$(t^{\sim}\, c_{Tick}) < maxTime \wedge$

$c'_{Tick} = t\,(t^{\sim}\, c_{Tick} + 1)$

$\vee$

$active(Tick) \wedge$

$op\ Reset \in i_{Tick}? \wedge$

$c'_{Tick} = t\,0$

$\vee$

$active(Tick) \wedge$

$op\ Pause/Play \in i_{Tick}? \wedge$

$c'_{Tick} = c_{Tick}$

$\vee$

$active(Tick) \wedge$

$op\ Tick \in i_{Tick}? \wedge$

$sPaused \in i_{Tick}? \cup (o_{Tick}! \cap \{sPlaying, sPaused\}) \wedge$

$c'_{Tick} = c_{Tick}$

$\vee$

$active(Tick) \wedge$

$op\ Tick \in i_{Tick}? \wedge$

$sPlaying \in i_{Tick}? \cup (o_{Tick}! \cap \{sPlaying, sPaused\}) \wedge$

$c_{Tick} = t\, maxTime \wedge$

$c'_{Tick} = c_{Tick}$

$\vee$

$\neg\, active(Tick) \wedge$

$c'_{Tick} = c_{Tick}$

$)$

$o_{Tick}! = \{\}$

L.3 Operation schemas for Pause/Play

The Pause/Play chart has four normal transitions and two Reset transitions that have been combined into one. In this section we will write each of these schemas, as well as the inactive schema and the schema that combines these together, just like the previous section.

δ_{ft}

PP_{Paused}
$PP'_{Playing}$
$i_{PP}? : \mathbb{P}\, in_{PP}$
$active_ : \mathbb{P}\, \mu_{State}$
$o_{PP}! : \mathbb{P}\, out_{PP}$

$active(PP)$
$op\ PausePlay \in i_{PP}?$
$o_{PP}! = \{\}$

δ_{tf}

$PP_{Playing}$
PP'_{Paused}
$i_{PP}? : \mathbb{P}\, in_{PP}$
$active_ : \mathbb{P}\, \mu_{State}$
$o_{PP}! : \mathbb{P}\, out_{PP}$

$active(PP)$
$op\ PausePlay \in i_{PP}?$
$o_{PP}! = \{\}$

δ_{tt}

$PP_{Playing}$

$PP'_{Playing}$

$i_{PP}? : \mathbb{P}\, in_{PP}$

$active_ : \mathbb{P}\, \mu_{State}$

$o_{PP}! : \mathbb{P}\, out_{PP}$

$active(PP)$

$op\ Tick \in i_{PP}?$

$o_{PP}! = \{Playing\}$

δ_{ff}

$PP_{Playing}$

$PP'_{Playing}$

$i_{PP}? : \mathbb{P}\, in_{PP}$

$active_ : \mathbb{P}\, \mu_{State}$

$o_{PP}! : \mathbb{P}\, out_{PP}$

$active(PP)$

$op\ Tick \in i_{PP}?$

$o_{PP}! = \{Paused\}$

δ_{resetp}

PP'_{Paused}

$i_{PP}? : \mathbb{P}\, in_{PP}$

$active_ : \mathbb{P}\, \mu_{State}$

$o_{PP}! : \mathbb{P}\, out_{PP}$

$active(PP)$

$op\ Reset \in i_{PP}?$

$o_{PP}! = \{\}$

$Inactive_{PP}$

$\Xi Chart_{PP}$

$i_{PP}? : \mathbb{P}\, in_{PP}$

$active_ : \mathbb{P}\, \mu_{State}$

$o_{PP}! : \mathbb{P}\, out_{PP}$

$\neg\ active(PP)$

$o_{PP}! = \{\}$

ϵ_{PP}

$\Delta Chart_{PP}$

$i_{PP}?, o_{PP}! : \mathbb{P}\, \mu_{Signal}$

$active_ : \mathbb{P}\, \mu_{State}$

$active(PP)$

$c'_{PP} = c_{PP}$

$o_{PP}! = \{\}$

$\neg\ (S_{f1} \wedge \rho(guard_1))$

$\neg\ (S_{f2} \wedge \rho(guard_2))$

$\vdots$

$$\delta_{PP} == \delta_{tf} \vee \delta_{ft} \vee \delta_{tt} \vee \delta_{ff} \vee \delta_{resetp} \vee Inactive_{PP} \vee \epsilon_{PP}$$

L.4 Composed Chart TPP

In this section we present the init, state, and transition schemas for the composed chart *TPP*. This is the composition of *TickChart* and *Pause/PlayChart*.

$states_{TPP} : \mathbb{P}\, \mu_{States}$

$in_{TPP} : \mathbb{P}\, \mu_{Signal}$

$out_{TPP} : \mathbb{P}\, \mu_{Signal}$

$\Psi : \mathbb{P}\, \mu_{Signal}$

$states_{TPP} = states_{Tick} \cup states_{PP}$

$in_{TPP} = in_{Tick} \cup in_{PP}$

$out_{TPP} = out_{Tick} \cup out_{PP}$

The chart and init schemas are simple as we just combine the Tick and Pause/Play chart schemas. This gives us a chart that has two current states and two initial states.

$Chart_{TPP}$

$$Chart_{Tick}$$
$$Chart_{PP}$$

$Init_{TPP}$

$$Init_{Tick}$$
$$Init_{PP}$$

The transition model is more complicated, as we need to hide the inputs and outputs of the component charts. $\delta_{Tick} \wedge \delta_{PP}$ can be seen in the final line of the constraints. This schema changes the current state of both lower level charts based on the input signals it receives.

δ_{TPP}

$$\Delta Chart_{TPP}$$
$$i_{TPP}? : \mathbb{P}\, in_{TPP}$$
$$active_ : \mathbb{P}\, \mu_{State}$$
$$o_{TPP}! : \mathbb{P}\, out_{TPP}$$

$$active(PP) \Leftrightarrow active(Tick)$$
$$\exists\, i_{PP}?, o_{PP}!, i_{Tick}?, o_{Tick}! : \mathbb{P}\, \mu_{Signal} \bullet$$
$$i_{PP}? = (i_{TPP}? \cup (o_{TPP}! \cap \{sPlaying, sPaused\})) \cap in_{PP} \wedge$$
$$i_{Tick}? = (i_{TPP}? \cup (o_{TPP}! \cap \{sPlaying, sPaused\})) \cap in_{Tick} \wedge$$
$$o_{TPP}! = o_{PP}! \cup o_{Tick}! \wedge$$
$$\delta_{Tick} \wedge \delta_{PP}$$

L.5 Hiding Operator

Now that we have a simplified model of the composed chart we can proceed to the hiding operator. The hiding operator hides input and output signals. In this example we are only inputting the operation name signals, not the feedback signals, and we are not outputting any signals to the enviroment. We will call this level of the schema SW and we need to create new axiomatic definitions as well as state, init and operation schema.

$$states_{SW} : \mathbb{P}\,\mu_{States}$$
$$in_{SW} : \mathbb{P}\,\mu_{Signal}$$
$$out_{SW} : \mathbb{P}\,\mu_{Signal}$$

$$states_{SW} = states_{TPP}$$
$$in_{SW} = in_{TPP} \setminus \{sPaused, sPlaying\}$$
$$out_{SW} = out_{TPP} \setminus \{sPaused, sPlaying\}$$

The chart and init schemas are not effected by hiding signals. The chart still has two current states and two initial states.

$$Chart_{SW} == Chart_{TPP}$$

$$Init_{SW} == Init_{TPP}$$

The transition schema replaces the input and output signals with sets that do not contain the signals *sPaused* and *sPlaying*.

δ_{SW}

$$\Delta Chart_{SW}$$
$$i_{SW}? : \mathbb{P}\,in_{SW}$$
$$active_ : \mathbb{P}\,\mu_{State}$$
$$o_{SW}! : \mathbb{P}\,out_{SW}$$

$$\exists\, i_{TPP}?, o_{TPP}! : \mathbb{P}\,\mu_{Signal} \bullet$$
$$i_{TPP}? = i_{SW}? \wedge$$
$$o_{SW}! = o_{TPP}! \cap out_{SW} \wedge$$
$$\delta_{TPP}$$

We can expand and simplify this schema into the following:

δ_{SW}

$$\Delta Chart_{SW}$$
$$i_{SW}? : \mathbb{P}\, in_{SW}$$
$$active_ : \mathbb{P}\, \mu_{State}$$
$$o_{SW}! : \mathbb{P}\, out_{SW}$$

$$active(PP) \Leftrightarrow active(Tick)$$
$$((active(Tick) \land active(PP) \land ($$
$$op\ Tick \in i_{SW}? \land$$
$$(t^{\sim}\ c_{Tick}) < maxTime \land$$
$$c'_{Tick} = t\,(t^{\sim}\ c_{Tick} + 1) \land c_{PP} = p\,\mathbf{true} \land c'_{PP} = p\,\mathbf{true} \lor$$
$$op\ Reset \in i_{SW}? \land$$
$$c'_{PP} = p\,\mathbf{false} \land c'_{Tick} = t\,0 \lor$$
$$op\ Pause/Play \in i_{SW}? \land$$
$$c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{true} \land c'_{PP} = p\,\mathbf{false} \lor$$
$$op\ Pause/Play \in i_{SW}? \land$$
$$c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{false} \land c'_{PP} = p\,\mathbf{true} \lor$$
$$op\ Tick \in i_{SW}? \land$$
$$c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{false} \land c'_{PP} = p\,\mathbf{false} \lor$$
$$op\ Tick \in i_{SW}? \land$$
$$c_{Tick} = t\,maxTime \land$$
$$c'_{Tick} = c_{Tick} \land c_{PP} = p\,\mathbf{true} \land c'_{PP} = p\,\mathbf{true}) \lor$$
$$\neg\, active(Tick) \land \neg\, active(PP) \land$$
$$c'_{Tick} = c_{Tick} \land c'_{PP} = c_{PP}$$
$$) \land$$
$$o_{SW}! = \{\}$$
$$)$$

L.6 Step semantics

We can now hide $active_$ to give the step semantics schema $SWSys$.

$$
\begin{array}{|l}
\hline
\textit{SWSys} \\
\Delta \mathit{Chart}_{SW} \\
i_{SW}? : \mathbb{P}\, \mathit{in}_{SW} \\
o_{SW}! : \mathbb{P}\, \mathit{out}_{SW} \\
\hline
\exists\, \mathit{active}_ : \mathbb{P}\, \mu_{State} \bullet \\
\mathit{active}(SW) \\
\delta_{SW} \\
\hline
\end{array}
$$

This expands into the schema we presented previously in section 8.6. We also previously showed how we can use the new operation operator to convert *SWSys* into operations that are conformal with the original specification. We will now investigate the applicability and correctness properties for *Tick* and *Pause/Play*. We will be using the same retrieve relation as before:

$$
\begin{array}{|l}
\hline
R \\
\mathit{Stopwatch} \\
\mathit{Chart}_{SW} \\
\hline
t^{\sim}\, c_{Tick} = \mathit{time} \wedge \\
(c_{PP} = p\, \mathbf{false} \wedge \mathit{playing} = \mathbf{false} \\
\vee\, c_{PP} = p\, \mathbf{true} \wedge \mathit{playing} = \mathbf{true}) \\
\hline
\end{array}
$$

The $\mu\, \mathit{Tick}$ schema increments the state of TickChart until it reaches $\mathit{maxTime}$. Note that it does not change the state of PausePlayChart.

$$
\begin{array}{|l}
\hline
\mu\, \mathit{Tick} \\
\Delta \mathit{Chart}_{SW} \\
\hline
(t^{\sim}\, c_{Tick}) < \mathit{maxTime} \wedge \\
c'_{Tick} = t\,(t^{\sim}\, c_{Tick} + 1) \wedge c_{PP} = p\, \mathbf{true} \wedge c'_{PP} = p\, \mathbf{true} \\
\vee \\
c'_{Tick} = c_{Tick} \wedge c_{PP} = p\, \mathbf{false} \wedge c'_{PP} = p\, \mathbf{false} \\
\vee \\
c_{Tick} = t\, \mathit{maxTime} \wedge \\
c'_{Tick} = c_{Tick} \wedge c_{PP} = p\, \mathbf{true} \wedge c'_{PP} = p\, \mathbf{true} \\
\hline
\end{array}
$$

We start by looking at the applicability property.

$\mathbf{pre}\, \mathit{Tick} \mathrel{\widehat{=}} [\mathit{Stopwatch} \mid \mathit{playing} = \mathbf{true} \wedge \mathit{time} + 1 \leq \mathit{maxTime}\, \vee$
$\mathit{playing} = \mathbf{false} \wedge \mathit{time} \leq \mathit{maxTime}]$

$\mathbf{pre}\, \mu\, \mathit{Tick} \mathrel{\widehat{=}} [\mathit{Chart}_{SW} \mid \mathbf{true}]$

While the abstract Tick operation is only defined while $time \leq maxTime$, $\mu\, Tick$ is a total operation. This is because we are using the do nothing interpretation. Because of this applicability is trivially true.

$$\forall\, AState;\ CState;\ \bullet\ \mathbf{pre}\, AOp_i \land R \Rightarrow \mathbf{pre}\, COp_i$$

$\equiv$(Substitution)

$$\forall\, Stopwatch;\ Chart_{SW};\ \bullet\ \mathbf{pre}\ Tick \land R \Rightarrow \mathbf{true}$$

$\equiv (P \Rightarrow \mathbf{true})$

$$\mathbf{true}$$

Correctness property:

$$\forall\, State;\ StateMC;\ StateMC';\ ?AOp_i\ \bullet$$
$$\mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow \exists\, State' \bullet R' \land AOp_i$$

$\equiv$(Substitution)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW}\ \bullet$$
$$\mathbf{pre}\ Tick \land R \land COp_i \Rightarrow \exists\, State' \bullet t^{\sim}\, c'_{Tick} = time' \land$$
$$(playing = \mathbf{true} \land time' = time + 1 \land c'_{PP} = p\, \mathbf{true} \lor$$
$$playing = \mathbf{false} \land time' = time \land c'_{PP} = p\, \mathbf{false}) \land playing = playing'$$

$\equiv$(One-Point Rule)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW}\ \bullet$$
$$\mathbf{pre}\ Tick \land R \land COp_i \Rightarrow t^{\sim}\, c'_{Tick} \leq maxTime \land$$
$$(playing = \mathbf{true} \land t^{\sim}\, c'_{Tick} = time + 1 \land c'_{PP} = p\, \mathbf{true} \lor$$
$$playing = \mathbf{false} \land t^{\sim}\, c'_{Tick} = time \land c'_{PP} = p\, \mathbf{false})$$

$\equiv$(Substitution)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet \mathbf{pre}\ Tick \land$$
$$t^{\sim}\, c_{Tick} = time \land ((t^{\sim}\, c_{Tick}) < maxTime \land$$
$$c'_{Tick} = t\,(t^{\sim}\, c_{Tick} + 1) \land c_{PP} = p\, \mathbf{true} \land c'_{PP} = p\, \mathbf{true} \land playing = \mathbf{true} \lor$$
$$c'_{Tick} = c_{Tick} \land c_{PP} = p\, \mathbf{false} \land c'_{PP} = p\, \mathbf{false} \land playing = \mathbf{false} \lor$$
$$c_{Tick} = t\, maxTime \land c'_{Tick} = c_{Tick} \land c_{PP} = p\, \mathbf{true} \land playing = \mathbf{true} \land$$
$$c'_{PP} = p\, \mathbf{true}) \Rightarrow t^{\sim}\, c'_{Tick} \leq maxTime \land$$
$$(playing = \mathbf{true} \land t^{\sim}\, c'_{Tick} = time + 1 \land c'_{PP} = p\, \mathbf{true} \lor$$
$$playing = \mathbf{false} \land t^{\sim}\, c'_{Tick} = time \land c'_{PP} = p\, \mathbf{false})$$

$\equiv (P \land Q \Rightarrow P)$

$$\mathbf{true}$$

Lastly we can show that the correctness property holds for Pause/Play. Applicability holds trivially because the μ-chart operation is total.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mu PP} \\
\hline
\Delta Chart_{SW} \\
\hline
c'_{Tick} = c_{Tick} \wedge c_{PP} = p\,\mathbf{true} \wedge c'_{PP} = p\,\mathbf{false} \vee \\
c'_{Tick} = c_{Tick} \wedge c_{PP} = p\,\mathbf{false} \wedge c'_{PP} = p\,\mathbf{true} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, State;\ StateMC;\ StateMC';\ ?AOp_i \bullet \\
\mathbf{pre}\, AOp_i \wedge R \wedge COp_i \Rightarrow \exists\, State' \bullet R' \wedge AOp_i
\end{array}
$$

$\equiv$(Substitution)

$$
\begin{array}{l}
\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet \mathbf{pre}\, AOp_i \wedge t^{\sim}\, c_{Tick} = time \wedge \\
(playing = \mathbf{true} \wedge c'_{Tick} = c_{Tick} \wedge c_{PP} = p\,\mathbf{true} \wedge c'_{PP} = p\,\mathbf{false} \vee \\
playing = \mathbf{false} \wedge c'_{Tick} = c_{Tick} \wedge c_{PP} = p\,\mathbf{false} \wedge c'_{PP} = p\,\mathbf{true}) \Rightarrow \\
\exists\, State' \bullet time = time' \wedge t^{\sim}\, c'_{Tick} = time' \wedge \\
(playing = \mathbf{true} \wedge playing' = \mathbf{false} \wedge c'_{PP} = p\,\mathbf{false} \vee \\
playing = \mathbf{false} \wedge playing' = \mathbf{true} \wedge c'_{PP} = p\,\mathbf{true})
\end{array}
$$

$\equiv$(One-point Rule)

$$
\begin{array}{l}
\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet \mathbf{pre}\, AOp_i \wedge t^{\sim}\, c_{Tick} = time \wedge \\
(playing = \mathbf{true} \wedge c'_{Tick} = c_{Tick} \wedge c_{PP} = p\,\mathbf{true} \wedge c'_{PP} = p\,\mathbf{false} \vee \\
playing = \mathbf{false} \wedge c'_{Tick} = c_{Tick} \wedge c_{PP} = p\,\mathbf{false} \wedge c'_{PP} = p\,\mathbf{true}) \Rightarrow \\
t^{\sim}\, c'_{Tick} = time \wedge \\
(playing = \mathbf{false} \wedge c'_{PP} = p\,\mathbf{true} \vee playing = \mathbf{true} \wedge c'_{PP} = p\,\mathbf{false})
\end{array}
$$

$\equiv (P \wedge Q \Rightarrow P)$

$$\mathbf{true}$$

Because all of the refinement properties hold we can conclude that this is a sound visualisation of the stopwatch specification.

Appendix M

Proofs for Figure 8.3

We also prove the soundness of Figure 8.3. This is also a visualisation of the stopwatch however it uses a local variable and assignment instead of chart composition to show the time. Below we present the schemas for this μ-chart after the operation operator has been applied. Note that we have changed the local variable name to avoid clashes while checking refinement holds.

These schemas have one current state, c_{SW}, and one local variable, $_time$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Chart}_{SW}} \\
\hline
c_{SW} : \textit{states}_{Stopwatch} \\
_time : \mathbb{N} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mu\,\textit{Init}} \\
\hline
\textit{Chart}'_{SW} \\
\hline
c'_{SW} = cPaused \\
_time' = 0 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mu\,\textit{Reset}} \\
\hline
\Delta\,\textit{Chart}_{SW} \\
\hline
c'_{SW} = cPaused \\
_time' = 0 \\
\hline
\end{array}$$

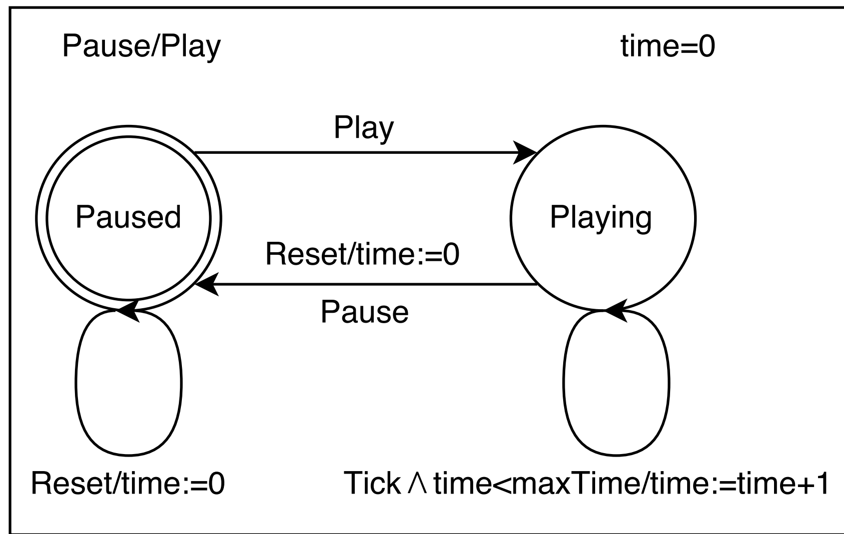
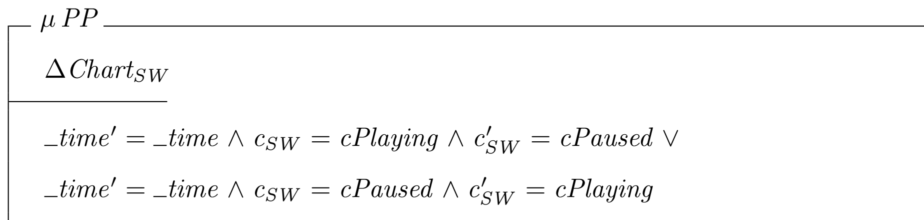


Figure 8.3: Microchart with Local Variable (repeated from page 95)

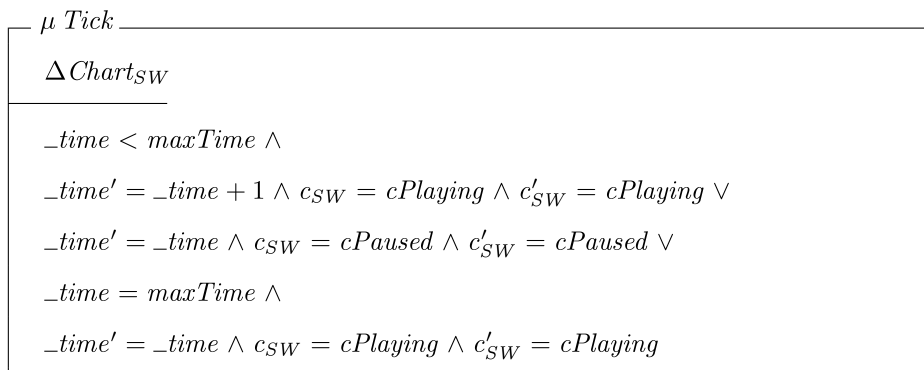


$\mu\, PP$

$\Delta Chart_{SW}$

$_time' = _time \land c_{SW} = cPlaying \land c'_{SW} = cPaused \lor$

$_time' = _time \land c_{SW} = cPaused \land c'_{SW} = cPlaying$



$\mu\, Tick$

$\Delta Chart_{SW}$

$_time < maxTime \land$

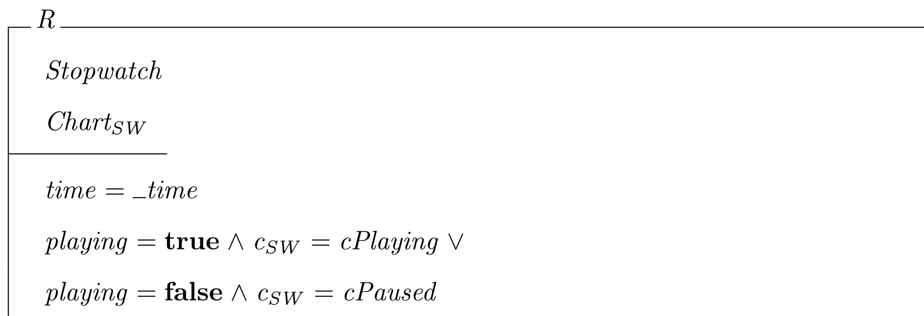
$_time' = _time + 1 \land c_{SW} = cPlaying \land c'_{SW} = cPlaying \lor$

$_time' = _time \land c_{SW} = cPaused \land c'_{SW} = cPaused \lor$

$_time = maxTime \land$

$_time' = _time \land c_{SW} = cPlaying \land c'_{SW} = cPlaying$

These are total operation schemas so proving applicability holds is trivial.



R

$Stopwatch$

$Chart_{SW}$

$time = _time$

$playing = \mathbf{true} \land c_{SW} = cPlaying \lor$

$playing = \mathbf{false} \land c_{SW} = cPaused$

We begin by checking that the initialisation property holds.

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \land R'$$

$\equiv$(Substitution)

$$\forall\, Chart'_{SW} \bullet c'_{SW} = cPaused \land _time' = 0 \Rightarrow$$
$$\exists\, time';\ playing' \bullet time' = 0 \land playing' = \mathbf{false} \land$$
$$time' = _time' \land c'_{SW} = cPaused$$

$\equiv$(One-point rule)

$$\forall\, Chart'_{SW} \bullet c'_{SW} = cPaused \land _time' = 0 \Rightarrow$$
$$0 = _time' \land c'_{SW} = cPaused$$

$\equiv$

$$\mathbf{true}$$

Then we can check that the correctness property holds for the three operations, starting with Pause/Play:

$$\forall\, State;\ StateMC;\ StateMC';\ ?AOp_i \bullet$$
$$\mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow \exists\, State' \bullet R' \land AOp_i$$

$\equiv$(Substitution)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet$$
$$R \land (_time' = _time \land c_{SW} = cPlaying \land c'_{SW} = cPaused \lor$$
$$_time' = _time \land c_{SW} = cPaused \land c'_{SW} = cPlaying) \Rightarrow$$
$$\exists\, State' \bullet R' \land time = time' \land \neg\, playing = playing'$$

$\equiv$(One-point rule)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet$$
$$R \land (_time' = _time \land c_{SW} = cPlaying \land c'_{SW} = cPaused \lor$$
$$_time' = _time \land c_{SW} = cPaused \land c'_{SW} = cPlaying) \Rightarrow$$
$$_time' = time \land (playing = \mathbf{false} \land c'_{SW} = cPlaying \lor$$
$$playing = \mathbf{true} \land c'_{SW} = cPaused)$$

$\equiv$(Expand R)

$$\mathbf{true}$$

Next we look at Reset:

$$\forall\, State;\ StateMC;\ StateMC';\ ?AOp_i \bullet$$
$$\mathbf{pre}\, AOp_i \land R \land COp_i \Rightarrow \exists\, State' \bullet R' \land AOp_i$$

$\equiv$(Substitution)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet$$
$$R \wedge c'_{SW} = cPaused \wedge _time' = 0 \Rightarrow$$
$$\exists\, State' \bullet time' = 0 \wedge playing' = \mathbf{false} \wedge$$
$$time' = _time' \wedge c'_{SW} = cPaused$$

$\equiv$(One-point rule)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet$$
$$R \wedge c'_{SW} = cPaused \wedge _time' = 0 \Rightarrow$$
$$0 = _time' \wedge c'_{SW} = cPaused$$

$\equiv$

$$\mathbf{true}$$

Finally, we look at Tick.

$$\forall\, State;\ StateMC;\ StateMC';\ ?AOp_i \bullet$$
$$\mathbf{pre}\, AOp_i \wedge R \wedge COp_i \Rightarrow \exists\, State' \bullet R' \wedge AOp_i$$

$\equiv$(Substitution)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet$$
$$(playing = \mathbf{true} \wedge time + 1 \leq maxTime \vee playing = \mathbf{false}) \wedge$$
$$R \wedge (_time < maxTime \wedge$$
$$_time' = _time + 1 \wedge c_{SW} = cPlaying \wedge c'_{SW} = cPlaying \vee$$
$$_time' = _time \wedge c_{SW} = cPaused \wedge c'_{SW} = cPaused \vee$$
$$_time = maxTime \wedge _time' = _time \wedge c_{SW} = cPlaying \wedge c'_{SW} = cPlaying)$$
$$\Rightarrow \exists\, State' \bullet time' \leq maxTime \wedge time' = _time' \wedge$$
$$(playing = \mathbf{true} \wedge c'_{SW} = cPlaying \wedge time' = time + 1 \vee$$
$$playing = \mathbf{false} \wedge c'_{SW} = cPaused \wedge time' = time) \wedge playing = playing'$$

$\equiv$(One-point rule)

$$\forall\, Stopwatch;\ Chart_{SW};\ Chart'_{SW} \bullet$$
$$(playing = \mathbf{true} \wedge time + 1 \leq maxTime \vee playing = \mathbf{false}) \wedge$$
$$R \wedge (_time < maxTime \wedge$$
$$_time' = _time + 1 \wedge c_{SW} = cPlaying \wedge c'_{SW} = cPlaying \vee$$
$$_time' = _time \wedge c_{SW} = cPaused \wedge c'_{SW} = cPaused \vee$$
$$_time = maxTime \wedge _time' = _time \wedge c_{SW} = cPlaying \wedge c'_{SW} = cPlaying)$$
$$\Rightarrow _time' \leq maxTime \wedge (playing = \mathbf{true} \wedge c'_{SW} = cPlaying \wedge _time' = time + 1 \vee$$
$$playing = \mathbf{false} \wedge c'_{SW} = cPaused \wedge _time' = time)$$

$\equiv$(Expand R)

true

So we conclude that this is a sound visualisation of the stopwatch specification.

Appendix N

Construction of a Sound Animation

In this example we construct a sound visualisation of the *Jars* specification. To do this we will use the animation function defined in section 5.5.

$$\begin{array}{l}
\textbf{R} \\
\hline
Level \\
animation_function : (\mathbb{N} \times Jars) \mapsto Images \\
\hline
animation_function = \\
\quad (\{ l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq max_fill\ c \bullet \\
\qquad global_maximum + 1 - l \mapsto c \} \times \{Empty\}) \oplus \\
\quad (\{ l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq level\ c \bullet \\
\qquad global_maximum + 1 - l \mapsto c \} \times \{Filled\})
\end{array}$$

This animation function will be our retrieve relation as it relates the specification state and animation state. This relation is functional from concrete to abstract so we can calculate our concrete operation and init schemas using the following:

$$CInit \mathrel{\widehat=} \exists\, AState' \bullet AInit \land R'$$

$$COp \mathrel{\widehat=} \exists\, \Delta State_A \bullet R \land AOp \land R'$$

This is a functional relation because every animation state matches one specification state. Figure 5.6 shows an example of the animation state where $j5$ is 3 units full. This matches the state where $level = \{(j3, 0), (j5, 3)\}$.

We begin with *CInit* which specifies the state that is initially shown by the animation. We expect this to be two empty jars of the correct size.

$$CInit \mathrel{\widehat=} [animation_function' \mid \exists\, level' \bullet \operatorname{ran} level' = \{0\} \land R']$$

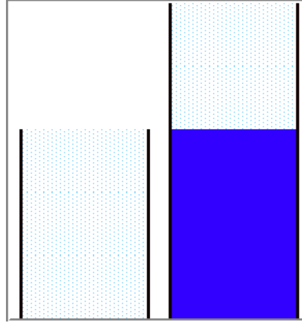


Figure 5.6: 2x5 Grid of Images (repeated from page 53)

≡(Substitution)

$$CInit \mathrel{\widehat=} [animation_function' \mid \exists\, level' \bullet \operatorname{ran} level' = \{0\} \land animation_function' =$$
$$(\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq max_fill\ c \bullet$$
$$global_maximum + 1 - l \mapsto c\} \times \{Empty\}) \oplus$$
$$(\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq level'\ c \bullet$$
$$global_maximum + 1 - l \mapsto c\} \times \{Filled\})]$$

≡(One-point rule)

$$CInit \mathrel{\widehat=} [animation_function' \mid animation_function' =$$
$$(\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq max_fill\ c \bullet$$
$$global_maximum + 1 - l \mapsto c\} \times \{Empty\}) \oplus$$
$$\varnothing]$$

This schema initialises the grid of images with *Empty* down from 5. This is upside down because the ProB animation function starts drawing from the top-left. The Z set-builder notation can be simplified further:

$$animation_function' = \{(5, j3), (4, j3), (3, j3), (5, j5), \ldots\} \times \{Empty\}$$

Next we calculate the *CFill* operation.

$$COp \mathrel{\widehat=} \exists\, \Delta State_A \bullet R \land AOp \land R'$$

$$CFill \mathrel{\widehat=} [\Delta animation_function;\ j? \mid \exists\, \Delta Level \bullet$$
$$R \land level(j?) < max_fill(j?) \land$$
$$level' = level \oplus \{j? \mapsto max_fill(j?)\} \land R']$$

$\equiv$(One-point rule R')

$$\begin{aligned}
&CFill \mathrel{\widehat{=}} [\Delta animation_function;\ j? \mid \exists\, level \bullet\\
&R \land level(j?) < max_fill(j?) \land animation_function' =\\
&\quad (\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq max_fill\ c \bullet\\
&\qquad global_maximum + 1 - l \mapsto c\} \times \{Empty\}) \oplus\\
&\quad (\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq (level \oplus \{j? \mapsto max_fill(j?)\})\ c \bullet\\
&\qquad global_maximum + 1 - l \mapsto c\} \times \{Filled\})]
\end{aligned}$$

$\equiv$(Substitution)

$$\begin{aligned}
&CFill \mathrel{\widehat{=}} [\Delta animation_function;\ j? : Jars \mid \exists\, level \bullet animation_function =\\
&\quad (\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq max_fill\ c \bullet\\
&\qquad global_maximum + 1 - l \mapsto c\} \times \{Empty\}) \oplus\\
&\quad (\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq level\ c \bullet\\
&\qquad global_maximum + 1 - l \mapsto c\} \times \{Filled\}) \land level(j?) < max_fill(j?)\\
&\land animation_function' =\\
&\quad (\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq max_fill\ c \bullet\\
&\qquad global_maximum + 1 - l \mapsto c\} \times \{Empty\}) \oplus\\
&\quad (\{l : 1 \mathrel{..} global_maximum;\ c : Jars \mid l \leq (level \oplus \{j? \mapsto max_fill(j?)\})\ c \bullet\\
&\qquad global_maximum + 1 - l \mapsto c\} \times \{Filled\})]
\end{aligned}$$

Unfortunately we cannot simply use the one-point rule to remove *level*. However, this is still a valid operation schema. The apparent complexity of this schema is due to two reasons. Firstly, there is only a single observation with a complex type. If this was split into multiple observations, such as an observation for each jar, then this would be much simpler. Secondly, this schema mostly consists of graphical information. Half of this schema just draws two empty jars. The fill operation here specifies how a particular layout of images changes when the operation is used. Despite the complexity however, this schema can be calculated by a computer and simply drawn as an animation rather than the user having to interface with the schema itself.