Continuous Probability Distributions in Model-Based Specification Languages

Thesis by

Zoe Helen Andrews

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at Newcastle University.

School of Computing Science,
Newcastle University,
Newcastle upon Tyne,
NE1 7RU,
UK.

August 2012

Abstract

Model-based specification languages provide a means for obtaining assurance of dependability of complex computer-based systems, but provide little support for modelling and analysing fault behaviour, which is inherently probabilistic in nature. In particular, the need for a detailed account of the role of continuous probability has been largely overlooked.

This thesis addresses the role of continuous probability in model-based specification languages. A model-based specification language (sGCL) that supports continuous probability distributions is defined. The use of sGCL and how it interacts with engineering practices is also explored. In addition, a refinement ordering for continuous probability distributions is given, and the challenge of combining non-determinism and continuous probability is discussed in depth.

The thesis is presented in three parts. The first uses two case studies to explore the use of probability in formal methods. The first case study, on flash memory, is used to present the capabilities of probabilistic formal methods and to determine the kinds of questions that require continuous probability distributions to answer. The second, on an emergency brake system, illustrates the strengths and weaknesses of existing languages and provides a basis for exploring a prototype language that includes continuous probability.

The second part of the thesis gives the formal definition of sGCL's syntax and semantics. The semantics is made up of two parts, the proof theory (transformer semantics) and the underpinning mathematics (relational semantics). The additional language constructs and semantical features required to include non-determinism as well as continuous probability are also discussed. The most challenging aspect lies in proving the consistency of the semantics when non-determinism is also included.

The third part uses a final case study, on an aeroplane pitch monitor, to demonstrate the use of sGCL. The new analysis techniques provided by sGCL, and how they fit in with engineering practices, are explored.

Declaration

I certify that no part of the material offered has been previously submitted by me for a degree or other qualification in this or any other University.

Published Work

Part of the work presented in this thesis has been published as follows.

1. A first version of the flash filestore case study presented in Chapter 3 appeared in:

   > Z.H. Andrews, A. McIver, L. Meinicke, and C. Morgan. Probabilistic aspects of flash filestores. In R. Joshi, T. Margaria, P. Müller, D. Naumann, and H. Yang, editors, *Int. Conf. on Verified Software: Theories, Tools and Experiments 2010, Workshop Proceedings*, 2010

   This research was carried out by Andrews with guidance from the remaining co-authors. In addition, jointly written background material from this paper is presented in Section 2.3.2.

2. A first version of the stochastic Event-B material presented in Chapter 4 appeared in:

   > Z.H. Andrews. Towards a stochastic Event-B. In *Supp. Volume of 2009 Workshop on Quantitative Formal Methods: Theory and Applications*, 2009

Acknowledgements

I would like to thank all the people who have provided me with valuable advice and support throughout my doctoral studies. In particular, I would like to thank my supervisor John Fitzgerald for his endless support, guidance and feedback on my research. I would also like to thank my thesis committee members Aad van Moorsel and Jason Steggles for their valued knowledge and advice.

During my studies I was grateful to have the opportunity to travel to Australia and work closely with Annabelle McIver, Carroll Morgan and Larissa Meinicke on the probabilistic aspects of flash filestores (see Chapter 3). I would like to thank them (and the Australian Research Council, and the School of Computer Science and Engineering at the University of New South Wales) for making this possible. I also appreciate the continued advice, particularly on the semantics of probabilistic formal methods, and support received from them.

I am grateful for the support received from my colleagues in the School of Computing Science at Newcastle University. Particular thanks go to Cliff Jones, Jeremy Bryans, Carl Gamble, Richard Payne, Ken Pierce and Claire Ingram, who have all at some point given me useful advice or feedback.

The research presented in this thesis would not have been possible without financial assistance. I am grateful to the EPSRC for providing a stipend and to the School of Computing Science at Newcastle University for choosing to support me financially. Many useful collaborations would not have been possible without the DEPLOY project.

Finally, I am grateful to my family for their endless support and encouragement. I am especially indebted to Will McElderry, who gave me the strength to persevere.

Contents

Abstract i

Declaration ii

Acknowledgements iii

Table of Contents iv

List of Figures ix

1 Introduction 1

1.1 Motivation and Problem Definition 1

1.1.1 Approach 2

1.2 Aims, Contribution and Methods 3

1.2.1 Aims 3

1.2.2 Contribution 4

1.2.3 Methods 5

1.3 Thesis Organisation 5

I Exploring Probability in Formal Methods 8

2 Background and Related Work 9

2.1 Probability Theory 9

2.1.1 Introducing probability 9

2.1.2 Probability distributions 10

2.1.3 Expectation and variance 12

2.1.4 Markov chains 13

2.2 Formal Methods 15

2.2.1 Model-based specification languages 16

2.2.2 Introducing GCL 17

2.3 Probability in Formal Methods 19

2.3.1 Probabilistic formal methods 20

2.3.2 Introducing pGCL 20
2.3.3 Stochastic formal methods 22
2.3.4 Probabilistic PDL 23

3 Probability in Formal Methods in Practice 25
3.1 Flash Filestore Systems 25
3.1.1 Wear-levelling 26
3.2 Specification of a Flash Filestore 27
3.2.1 Modelling a probabilistic wear-levelling algorithm in pGCL . . . 28
3.2.2 Analysis 28
3.3 Continuous Probability in Flash Filestores 31
3.3.1 When does a flash filestore fail? 31
3.3.2 Considering trade-offs 33
3.4 Applications of Continuous Probability in Computing 34
3.5 Concluding Remarks 35

4 Towards a Stochastic Model-Based Specification Language 36
4.1 The Emergency Brake 37
4.2 Modelling the Emergency Brake in PRISM 38
4.2.1 PRISM overview 39
4.2.2 PRISM models and analysis 39
4.2.3 Experiences with PRISM 41
4.3 Modelling the Emergency Brake in pB 42
4.3.1 pB overview 42
4.3.2 Discrete approximations of the emergency brake system 44
4.3.3 pB models and analysis 45
4.3.4 Experiences with pB 48
4.4 Modelling the Emergency Brake in Stochastic Event-B 49
4.4.1 Event-B overview 49
4.4.2 Stochastic extensions 50
4.4.3 Event-B models and analysis 51
4.4.4 Experiences with Stochastic Event-B 54
4.5 Next Steps and Challenges 56
4.6 Concluding Remarks 57

II Defining sGCL 58

5 Foundations of sGCL 59
5.1 Measure Theory 59
5.1.1 Basic definitions 59
5.1.2 Integration over measures 60

5.2 A Stochastic Powerdomain 62
5.3 A Metric Space for Measures 64
5.3.1 Joint probability distributions 64
5.3.2 The Kantorovich metric 65
5.4 Semantics of pGCL 66
5.4.1 Syntax review 67
5.4.2 Transformer semantics 68
5.4.3 Relational semantics 69
5.4.4 Relating the transformer and relational semantics 74
5.5 Concluding remarks 75

6 A Deterministic sGCL **76**
6.1 Preliminaries 76
6.2 Syntax 77
6.3 Transformer Semantics 79
6.3.1 Healthiness of the transformer semantics 81
6.4 Relational Semantics 89
6.4.1 Linking the transformer and relational semantics 90
6.5 Refinement Notions 94
6.5.1 Reducing non-termination 94
6.5.2 Data refinement 95
6.6 Discussion 96

7 Towards a Non-Deterministic sGCL **98**
7.1 Syntax and Transformer Semantics 98
7.1.1 Healthiness conditions 99
7.2 Relational Semantics 100
7.2.1 Deterministic relational semantics 100
7.2.2 Non-deterministic relational semantics 103
7.3 Relating the Transformer and Relational Semantics 106
7.3.1 Proving the consistency of the two semantics 107
7.4 Refinement Notions 108
7.4.1 Reducing non-determinism 108
7.5 Discussion 110

III Applying and Evaluating sGCL **112**

8 Applying sGCL: Patterns and Pitch **113**
8.1 Design Patterns 113
8.2 The Monitoring Voter in sGCL 117
8.3 The Aeroplane Pitch Monitor 123

8.3.1 Overview of the pitch monitor . . . 123
8.3.2 Modelling and analysing the pitch monitor in sGCL . . . 125
8.4 Further Exploration . . . 129
8.4.1 Adding a third sensor . . . 129
8.4.2 Sensor refinement . . . 133
8.5 Evaluation . . . 136

9 Conclusions 138
9.1 Summary . . . 138
9.2 Evaluation . . . 140
9.2.1 General aims . . . 140
9.2.2 Language aims . . . 143
9.2.3 Discussion of the fault classes sGCL may apply to . . . 145
9.3 Further Work . . . 146
9.3.1 Language extensions . . . 146
9.3.2 Methodological extensions . . . 148
9.3.3 Tool support . . . 149
9.4 Closing statements . . . 150

Appendix A Proving the Flash Memory Loop Invariant 151

Appendix B Emergency Brake Models and Proofs 157
B.1 PRISM Models . . . 157
B.1.1 Version 1 . . . 157
B.1.2 Version 2 . . . 158
B.2 pB Models . . . 160
B.2.1 Option 1 . . . 160
B.2.2 Option 2 . . . 161
B.3 pB Expectation Analysis . . . 162
B.3.1 Option 1 . . . 162
B.3.2 Option 2 . . . 165
B.4 Event-B Models . . . 170
B.4.1 Standard Event-B . . . 170
B.4.2 Stochastic Event-B option 1 . . . 171
B.4.3 Stochastic Event-B option 2 . . . 173
B.5 Stochastic Event-B Expectation Analysis . . . 174
B.5.1 Option 1 . . . 174
B.5.2 Option 2 . . . 175

Appendix C Proving the Healthiness Conditions in Deterministic sGCL 180
C.1 Continuity . . . 180

C.2 Linearity 183

Appendix D The Challenge of Proving Consistency in a Non-Deterministic sGCL **188**

D.1 Consistency Proofs 188

D.1.1 A discrete approximation for measures 189

D.1.2 The challenge of showing that *wp* is an injection 193

D.1.3 The challenge of showing that *rp* is an injection 200

D.2 A Note on the Use of the Kantorovich Metric with Sub-Probability Measures 203

Appendix E Supplementary Lemmas **205**

Appendix F DESTECS Patterns **207**

F.1 Voter Pattern 207

F.2 Monitor 208

Bibliography **210**

List of Figures

2.1 Probability mass functions of various discrete probability distributions . 11
2.2 Probability density functions of various continuous probability distributions 13
2.3 Example Markov chain with three states 14
2.4 Notation and weakest pre-condition semantics of GCL 19
2.5 Notation and weakest pre-condition semantics of pGCL [6] 21
2.6 Notation and expectation transformer semantics of probabilistic PDL . 24

3.1 Abstract representation of a flash filestore garbage collector 27
3.2 Specification of the probabilistic wear-levelling algorithm 28
3.3 Expected lifetime ($E[e]$) for the probabilistic wear-levelling algorithm . . 31

4.1 States and transitions for the EB system 38
4.2 PRISM syntax for CTMC models 40
4.3 First PRISM model - unsafe failure event 41
4.4 Second PRISM model - synchronised request event 41
4.5 Sample results of the emergency brake analysis in PRISM 42
4.6 Notation and expectation transformer semantics of pGSL 43
4.7 pB *main* and *EB_Request* operations for option 1 46
4.8 pB *main* and *EB_Request* operations for option 2 47
4.9 Stochastic Event-B *EB_Request* event for option 1 52
4.10 Stochastic Event-B *EB_Request* event for option 2 52

5.1 Monad coherence conditions represented as commutative diagrams [66] . 63
5.2 Review of notation and weakest-precondition semantics of pGCL [6] . . 67
5.3 Healthiness conditions for pGCL transformers. 69

6.1 Syntax and weakest pre-condition semantics of deterministic sGCL . . . 80
6.2 Examples of measure ordering: mu 1 (solid line); mu 2 (dashed line) . . 91

7.1 Syntax and weakest pre-condition semantics of non-deterministic sGCL 99

8.1 Class (left) and object (right) diagrams for the monitoring voter pattern 116
8.2 A monitoring voter with two replicated sensors 118

8.3 A monitoring voter with two diverse sensors 120
8.4 The probability of triggering a reset action for a variety of sensor pairs . 123
8.5 Probability of a reset action against tolerance for various sensors 124
8.6 Aeroplane attitudes and axes (left) and pitch monitor state chart (right) 125
8.7 Specification of the pitch monitor 126
8.8 A monitoring voter with three replicated sensors 130
8.9 Probability of a reset action for three (solid) and two (dashed) sensors . 133
8.10 The cross-section of the volume for the different three sensor strategies . 134
8.11 The probability density functions of mu (solid) and mu' (dashed) 135

D.1 The ϵ-approximation (solid) of a truncated exponential distribution (dashed) 190

F.1 Voter pattern (class and object diagram) 208
F.2 Monitor pattern (class and object diagram) 209

Chapter 1

Introduction

This chapter motivates the work presented in this thesis and describes the problem that it aims to solve. The contribution of the thesis is presented before detailing the methods used to carry out the research. Finally the structure of the rest of the document is summarised.

1.1 Motivation and Problem Definition

The range and complexity of systems in which computers play a major part is ever increasing. As complexity increases it becomes harder to design computer-based systems that function predictably. At the same time, greater reliance is being placed on their correct functioning, in particular in those systems that are responsible for preserving lives or livelihood. Providing the required increases in dependability is only possible if the techniques for obtaining and *assuring* such dependability are continually improved and updated.

One approach to providing assurance about system properties is through the use of "model-based specification languages". *Specification languages* are equipped with abstractions for expressing system properties with minimal bias towards the eventual system design or implementation. *Formal specification languages* have mathematically defined semantics, which provide the required assurance of system properties through formal validation and verification techniques. *Model-based (formal) specification languages* use an abstract system model to communicate the desired behaviour (as opposed to listing properties as axioms). Such languages are intended to be more accessible to engineers than their underlying mathematics, thus enabling more widespread use of rigorous analysis of system properties.

To model and analyse the dependability of a system, its fault behaviour, as well as normal operation, needs to be considered. Faults often have a random nature that can only be captured by probability. There is currently limited support for probability in model-based specification languages. In particular, the support for *continuous* prob-

ability is largely unexplored in these languages. A *continuous probability distribution* describes the probability of a range of values, as opposed to a set of values. Such distributions are used throughout engineering, for example, in modelling the expected time between faults.

The problem explored in this thesis is how to incorporate continuous probability into model-based (formal) specification languages, in particular for the analysis of the fault behaviour of computer-based systems.

1.1.1 Approach

The approach taken in this thesis draws from two key research areas in computing: "Dependability" and "Formal Methods". These are two fairly distinct research communities with a common goal – to empower more reliable computer-based systems.

Dependability is defined by Avizienis et al. [9] as "the ability to deliver service that can justifiably be trusted". The same authors identify four means to attain dependability: fault prevention; fault tolerance; fault removal; and fault forecasting. In practice, dependability research tends to focus on tolerating and forecasting faults. The features of dependability research that this thesis makes use of are fault tolerant architectures and the quantitative evaluation of dependability attributes. Fault tolerant architectures make use of redundancy to reduce the impact of faults occurring on the overall system [59]. The main dependability attribute that will be analysed is "reliability" (the continuity of correct service [9]), although "safety" (the absence of catastrophic consequences on the user(s) and the environment) is also considered.

Formal methods use "mathematically-based languages, techniques and tools for specifying and verifying computer-based systems" [16]. Note that some would consider formal methods a research area within dependability, falling under the fault prevention category. In practice, the two research communities have little overlap. The main approach drawn from formal methods in this thesis is the development of a model-based specification language and its semantics. In addition, the thesis is concerned with the verification approach of "theorem proving" and a development approach known as "refinement". *Theorem proving* is the verification approach that the language developed in this thesis is intended to support. It is defined as the process of finding a proof of a property from the axioms of the system [16]. *Refinement*, also supported by the language described in this thesis, plays an important role in the use of formal methods as a stepwise development process. The refinement process allows an abstract model to be produced and key properties of it verified, before more details are added in a series of refinement steps. Proofs are used to show that the refined models respect the abstraction and thus that the key properties are preserved [44].

Formal techniques are a promising approach to developing dependable systems because they allow the interaction between desired behaviour and dependability to be evaluated rigorously. However, without support for continuous probability distribu-

tions, the frequency and impact of randomly occurring faults are difficult to express and analyse. The approach taken in this thesis is to extend a formal method with the ability to express continuous probability. This allows dependability properties (such as reliability) to be treated as rigorously as standard (functional) properties. This means that:

- the claimed benefits of fault tolerant architectures can be verified formally;
- the dependability of differing designs can be compared accurately;
- the relationship between program parameters affecting overall dependability can be explored.

Case studies are used to develop the new language and evaluate its success at achieving the above features.

1.2 Aims, Contribution and Methods

1.2.1 Aims

The overall vision of this research is to equip developers with the ability to reason about systems with (continuous) probabilistic elements, in the context of model-based formal methods for developing dependable systems. In order to do that, it is required to satisfy the following general aims:

- to explore the role of probability in model-based specification languages, in particular continuous probability;
- to develop a model-based specification language that supports reasoning about continuous probability;
- to demonstrate reasoning about dependability properties using the new language;
- to examine how the new language fits into software engineering practice, such as the use of design patterns.

The language that will be developed is a key feature of the thesis, therefore this is examined more closely. The new language should:

- include explicit support for continuous probability distributions, so that variables may be assigned values according to a continuous probability distribution;
- allow formal analysis and proof of (probabilistic) properties over systems containing continuous probability distributions, for example dependability attributes such as reliability and safety;

- provide means for abstraction and refinement of computer-based systems containing continuous probability distributions;
- be underpinned by a consistent mathematical foundation.

These are all important properties of model-based formal specification languages. In addition, it is desirable for the new language to:

- support non-determinism, where the program can branch according to some external decisions beyond the control of the program.

This provides enhanced abstraction capabilities that are particularly useful in the early stages of design.

1.2.2 Contribution

The contribution provided by this thesis is the exploration of the role of continuous probability distributions in model-based specification languages. This can be decomposed into three main aspects: the initial exploration of uses of continuous probability in formal methods; the development of a model-based specification language that supports continuous probability distributions; and the application of the new language from an engineering perspective.

The initial exploration demonstrates the use of probability in formal methods and determines the questions that are more suited to continuous probability. A new language is also prototyped based on the strengths and limitations found in the existing use of probability in formal methods. The complexities of formally describing this language in full are discussed.

The major part of the contribution is the development of the syntax and formal semantics of a model-based specification language, sGCL. The chosen base language is simpler than that of the prototype, to allow a formal description of the syntax, proof-theory and relational semantics. One key simplification made is that non-determinism is removed from the fully proven language, although the complexity of including both continuous probability and demonic non-determinism is discussed at length and suggestions are made for further work in this area. Nonetheless, the deterministic version of sGCL presented is designed to satisfy the essential properties for model-based specification languages as described above. Important healthiness conditions (properties that must hold to ensure that programs have meanings) for probabilistic programs are also shown to hold in sGCL. One particularly novel aspect of the language is the definition of a refinement relation between continuous probability distributions.

The final aspect of the contribution is methodological. The deterministic sGCL is shown to have the capability to analyse interesting dependability properties and fit in well with the engineering practice of design patterns.

1.2.3 Methods

In order to satisfy the aims of the thesis: a suitable base language, to which continuous probability can be added, must be identified; the extensions to the base language need to be defined and the semantics of the extended language specified; and the new language needs to be applied and evaluated.

This led to three main phases of research: first to examine the use of probability in formal methods and identify a suitable language to extend with continuous probability; second to develop a model-based specification language that supports continuous probability; third to explore and evaluate the application of the language developed.

The methods used in these phases include: language definition; specification of semantics; and the development of case studies for the purposes of language exploration and evaluation.

The first phase explored the use of probability in formal methods through the use of case studies. The case studies were used for two main purposes. The first, on flash memory, was used to understand the capabilities of probabilistic formal methods and to determine the kinds of questions that require continuous probability distributions to answer. The second, on an emergency brake system, was used to illustrate the strengths and weaknesses of existing languages and to explore a prototype language that includes continuous probability. The first phase concluded with the identification of a suitable language to be extended formally with continuous probability.

The second phase was the formal development of the new language, sGCL. This involved the definition of the language syntax and the specification of its semantics. The semantics is made up of two parts, the proof theory (transformer semantics) and the underpinning mathematics (relational semantics). The phase ended with an exploration of the additional language constructs and semantical revisions required to include nondeterminism, for the purposes of abstraction.

The third phase used a final case study to evaluate the usefulness of sGCL. The case study, on an aeroplane pitch monitor, was used to explore the new analysis techniques provided by sGCL and how such analysis fits in with engineering practices. The phase ended with a more general evaluation of the effectiveness of sGCL and the opportunities for further development of the research.

1.3 Thesis Organisation

The remainder of the thesis is structured into three parts. The first part contains background material on the use of probability in formal methods. The second part describes the new language, sGCL, detailing its syntax and semantics. The third part analyses the application of sGCL, evaluates its usefulness and identifies areas for further research.

The first part opens with an introduction to the basic concepts and definitions that

occur when discussing probability in formal methods (Chapter 2). This includes: an introduction to probability, and what is meant by continuous probability; followed by an overview of formal methods, particularly focussing on model-based specification languages. The existing approaches to extending formal methods with probability are then discussed, this includes an overview of the related work. In particular, the languages on which the majority of this thesis is based (GCL and pGCL) are summarised.

The first part then proceeds with a demonstration of the use of probability in model-based specification languages in practice (Chapter 3). This uses a case study on flash memory to show how probabilistic programs can be modelled and reasoned about in pGCL. This leads into a discussion of the kinds of questions that are more easily answered using continuous probability.

The conclusion of the first part of this thesis is an exploration of probability in formal methods more generally and the prototyping of a new approach (Chapter 4). A case study on an emergency brake is used to illustrate the strengths and limitations of existing approaches. A new language (that aims to combine the strengths of several existing approaches) is then prototyped and applied to the case study to demonstrate its use. The complexities involved in defining a full formal definition of the prototype language are discussed. This allows suitable simplifications to be identified for the language whose development is detailed in full in Part 2.

Part 2 starts with more technical background material (Chapter 5) that is required to understand the semantics of the new language. This includes an introduction to measure theory, as this is the basis of the underpinning mathematics of sGCL. Other topics introduced in this chapter include: a stochastic powerdomain for sequential composition of programs containing continuous probability; and a metric space for measures. The metric space is used for the discussion of a non-deterministic sGCL, where the proofs require reasoning about compactness of sets of measures. The chapter ends with a detailed look at the semantics of pGCL [61], as this provides the framework for the semantics of sGCL.

The main chapter in Part 2 is the description of sGCL (Chapter 6). In this chapter the syntax, proof theory and semantics are laid out for the language, sGCL, that constitutes the main contribution of this thesis. Vital healthiness conditions for probabilistic programs are shown to hold in sGCL. Refinement in sGCL is also discussed, including formal proof that the refinement orderings given in the proof-theory and relational semantics are consistent. The chapter ends with an evaluation and a discussion of the novelty of sGCL.

The language defined in Chapter 6 does not include support for demonic non-determinism. The final chapter (Chapter 7) of part two explores extensions to this language to include such non-determinism. This includes the definition of the necessary syntactic, proof-theoretic and semantic changes to sGCL. The challenge of proving the consistency of the proof theory and the relational semantics (when non-determinism is

introduced) is discussed briefly (a deeper discussion can be found in Appendix D). Finally the additional opportunities for refinement, when non-determinism is added, are explored.

Part 3 starts with an exploration of how sGCL can be applied in practice (Chapter 8). The engineering practice of design patterns is discussed and a new design pattern relating to dependability is described. The probabilistic aspects of this pattern are then analysed in sGCL before the pattern is applied in a final case study on an aeroplane pitch monitor. The opportunities for refinement in the case study are discussed, as is the potential for further analysis. Finally, the case study, and the performance of sGCL in analysing it, are evaluated.

Part 3, and the thesis, ends with a summary of the research carried out and a forward look (Chapter 9). The extent to which the aims of the thesis have been met is evaluated. This assessment leads into a discussion of the opportunities for further development of sGCL and its application to developing dependable systems.

Part I

Exploring Probability in Formal Methods

Chapter 2

Background and Related Work

This chapter sets the scene for the main contributions of the thesis, by presenting background material on probability, formal methods, and the state of the art in combining the two. This is intended to be a fairly lightweight introduction to these areas. The topics that require a deeper examination in order to define a formal language that incorporates continuous probability are described in more detail in Chapter 5.

2.1 Probability Theory

This thesis is concerned with modelling and analysing computer-based systems that incorporate some element of chance. Such modelling and analysis requires an elementary understanding of probability theory. This section introduces the terminology and basic concepts of probability theory that are required to understand this thesis. A more thorough treatment of probability can be found in many statistical and engineering textbooks [71] [55].

2.1.1 Introducing probability

Probability theory is relevant to any situation where an element of chance or *randomness* is possible. The process of observing or measuring the results of some random occurrence is known as an *experiment*. The possible results of the experiment are known as *outcomes*. The *sample space* is the set of possible outcomes of an experiment. Any subset of the sample space is called an *event*.

For example, an experiment could be the process of rolling a six-sided die. This process results in an integer between one and six – these are the outcomes. The sample space would be the set $\{1, 2, 3, 4, 5, 6\}$. Example events of rolling the die are the sets $\{6\}$ and $\{1, 2\}$.

Each event in an experiment has some *probability* of occurring. A probability can take any value in $[0, 1]$, where an event with probability zero is *impossible* and one with probability one is *certain*. The sample space of an experiment has probability one by

definition.

For example, assuming the six-sided die described above is *fair* (each outcome is equally likely), the probability of obtaining the event $\{6\}$, written $P(\{6\})$ is $\frac{1}{6}$. Similarly $P(\{1,2\})$ is $\frac{1}{3}$ as the event $\{1,2\}$ is interpreted as *either* a one or a two is thrown.

A *random variable* is any function from the sample space into the real numbers, or more intuitively a random number. If the sample space is finite or countably infinite the random variable is called *discrete*, otherwise it is called *continuous*.

For example, a discrete random variable may be the number of sixes thrown on a fair die in ten consecutive rolls or the number of consecutive rolls occurring before a six is thrown. A continuous random variable could be the time until the next bus arrives or the rainfall in mm at some specified city on a given day.

2.1.2 Probability distributions

A probability distribution defines the probability (or probability density) of the possible values of a random variable. If the random variable is discrete it follows a *discrete probability distribution*, otherwise if it is continuous it follows a *continuous probability distribution*. Therefore, discrete distributions usually range over (a subset of) the natural numbers, and continuous distributions over (a subset of) the real numbers.

Discrete probability distributions

Discrete probability distributions are defined using a *probability mass function* (or pmf), written $f(x)$, as follows

$$f(x) = \begin{cases} p_j & , \text{ if } x = x_j \\ 0 & , \text{ otherwise} \end{cases}$$

which specifies the probability of each possible value occurring.

For example the pmf of a negative binomial distribution is

$$f(x) = \begin{cases} \begin{pmatrix} x-1 \\ x-r \end{pmatrix} p^r (1-p)^{x-r} & , \ x = r, r+1, r+2, \ldots \\ 0 & , \text{ otherwise} \end{cases}$$

for some probability p and required number of successes (or failures) r. The negative binomial distribution models the number of trials observed until r instances of an event of interest have occurred, assuming a constant probability p of the event of interest occurring in each trial.

The *cumulative distribution function* (or cdf), written $F(x)$, can also be recorded in a similar way and specifies the probability of observing a value no greater than the input value x.

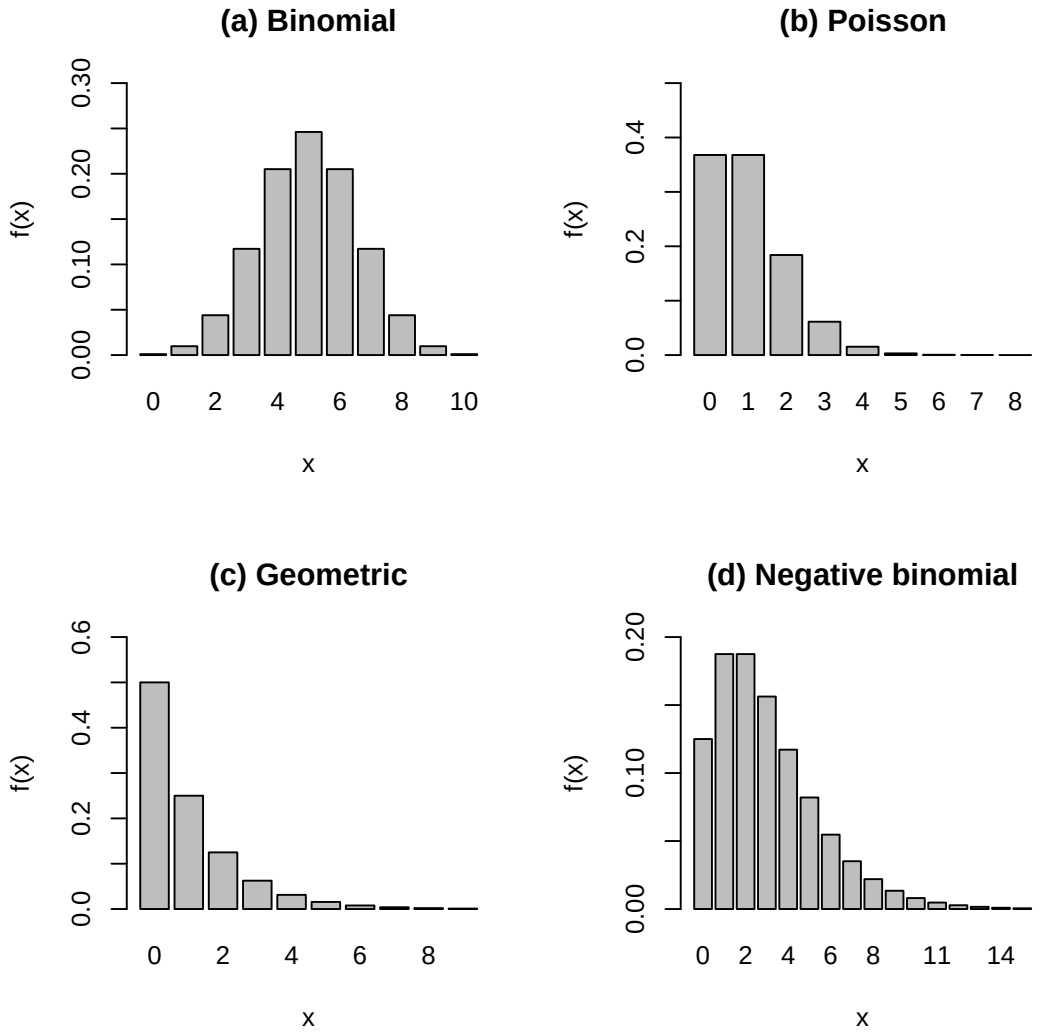

Figure 2.1: Probability mass functions of various discrete probability distributions

Frequently occurring discrete probability distributions include the binomial, Poisson, geometric and negative binomial [71]. Illustrative probability mass functions of these distributions are shown in Figure 2.1.

Continuous probability distributions

The definition for continuous probability distributions is slightly less intuitive. Continuous random variables have an infinite sample space, so it no longer makes sense to assign a probability to each outcome (any probability greater than zero for individual outcomes would clearly lead to an infinite probability being assigned to the entire sample space). Instead probabilities are assigned to subsets of the sample space. This results in a continuous function over (a subset of) the real numbers known as a *probability density function* (or pdf), written $f(x)$. The pdf is described such that

$$P(a < X \leq b) = \int_a^b f(x)dx$$

holds for all $a < b$.

For example the pdf of an exponential distribution is

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & , \ x \geq 0 \\ 0 & , \ \text{otherwise} \end{cases}$$

for some rate λ. The exponential distribution is commonly used to model the inter-arrival times between events of interest, where λ represents the arrival rate.

The *cumulative distribution function* (or cdf) of a continuous probability distribution, is again written $F(x)$ and has the same interpretation as with discrete distributions. It can be found by integrating the pdf up to x.

Frequently occurring continuous probability distributions include the normal (or Gaussian), exponential, hyper-exponential, uniform, chi-squared and gamma [71]. Illustrative probability density functions of continuous probability distributions are shown in Figure 2.2.

2.1.3 Expectation and variance

When analysing probabilistic systems two metrics are of particular interest, the *expectation* (or expected value) and the *variance* of a random variable.

The expectation of a random variable X, written $E[X]$, is the average value the random variable could take, weighted by the probability of it doing so. If X follows a discrete distribution the expectation of X is

$$E[X] = \sum_i x_i f(x_i)$$

Whilst if X follows a continuous distribution the expectation of X is

$$E[X] = \int_{-\infty}^{\infty} x f(x)$$

Note that x may be replaced with $g(x)$ in both of the above to find $E[g(X)]$, the expected value of some (non-constant and continuous) function g of the random variable X.

The variance of random variable X, written *Var(X)*, provides a measure of the range of values that X can take and is defined as

$$Var(X) = E[X^2] - E[X]^2$$

Taking the square root of the variance gives the *standard deviation* of the random variable, that is the average distance of the values from the expected value of the random variable.

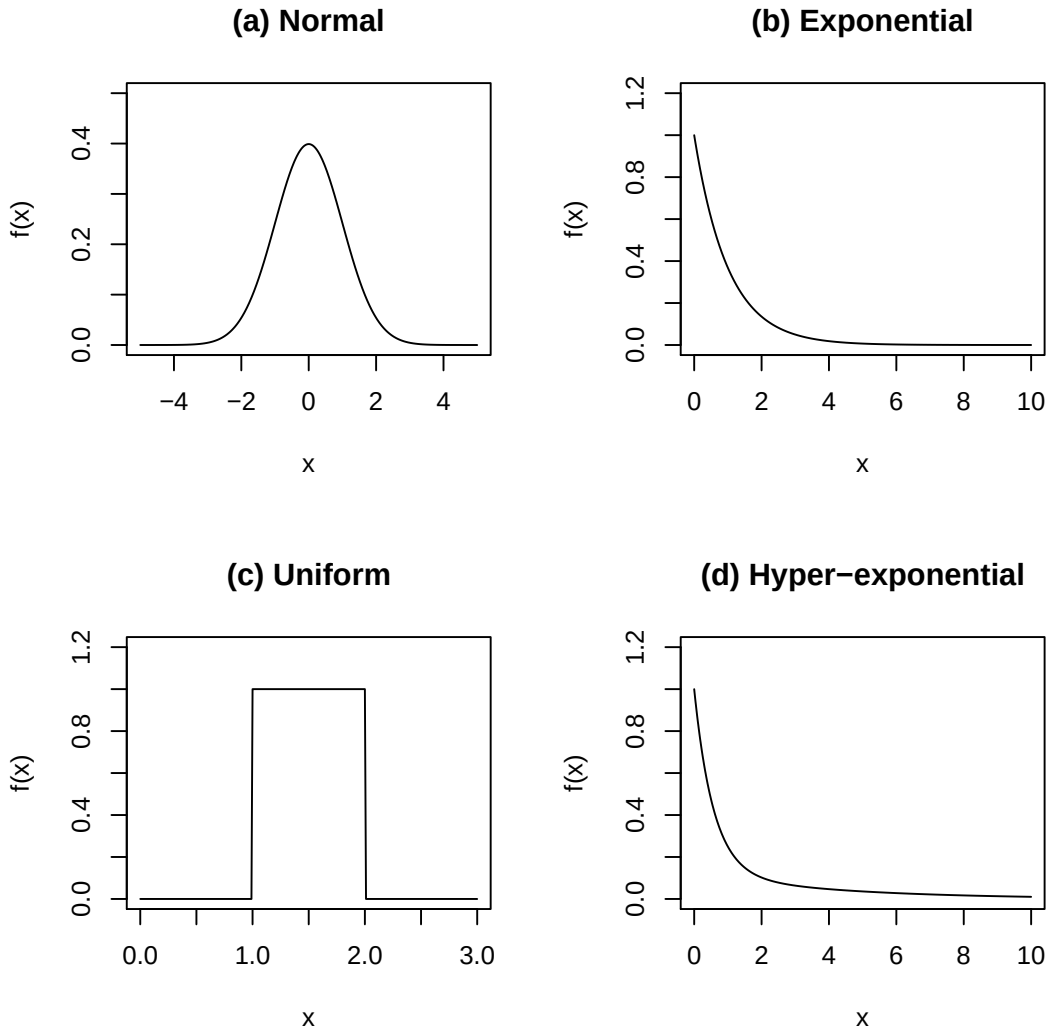

Figure 2.2: Probability density functions of various continuous probability distributions

2.1.4 Markov chains

This thesis is not directly concerned with the use of Markov Chains, however, a brief introduction is prudent in order to understand how the work presented in this thesis relates to other research.

A Markov chain consists of a (finite or countable) set of states and a set of possible transitions between these states. Each transition occurs with some fixed probability, or rate in the case of "CTMCs" (described below). A key feature of a Markov chain is that it is memoryless. This means that the probability of being in a given state after the next transition is only dependant on the current state – the history is irrelevant.

Markov chains may be represented diagrammatically or through the use of a transition matrix. An example, with three states and four transitions, is used to illustrate these representations. The diagrammatic representation of the example is shown in Fig-

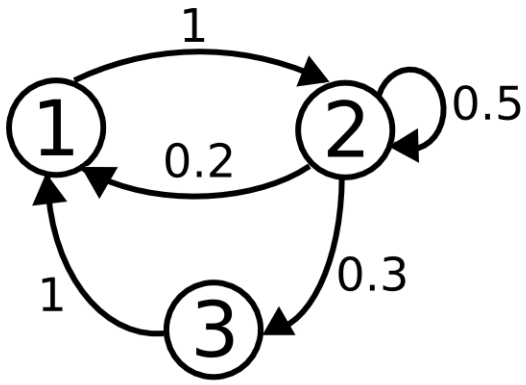

Figure 2.3: Example Markov chain with three states

ure 2.3. The circles represent the states whilst the arrows represent transitions between states, with the arrowhead indicating direction. The label on an arrow represents the probability of the transition occurring from the given starting state. The same example is recorded in a transition matrix as follows

$$\begin{pmatrix} 0 & 1 & 0 \\ 0.2 & 0.5 & 0.3 \\ 1 & 0 & 0 \end{pmatrix},$$

where p_{ij}, the probability of a transition occurring from state i to state j, is given in the i^{th} row and the j^{th} column of the matrix.

There are various properties that can be analysed using Markov chains, such as the probability of being in state x, after n time units, from a given starting state. However, the most interesting properties are those that relate to the *steady state* of a Markov chain. Certain classes of Markov chain enter a steady state as time approaches infinity. For these classes of Markov chains, a stationary distribution can be calculated that gives the probability of being in each state after a sufficiently long period of time has elapsed. This distribution can then be used to calculate properties such as the expected amount of time the Markov chain will spend in a specific state. This is clearly useful when certain states are more desirable than others.

There are two kinds of Markov chains that are discussed in this thesis. The first is as described above, the Discrete Time Markov Chain (DTMC). This treats time implicitly, state transitions occur at intervals exactly one time unit apart. The other kind of Markov chain is the Continuous Time Markov Chain (CTMC). This associates transitions with rates instead of probabilities (usually written λ_{ij} instead of p_{ij}). The timing of a transition in a CTMC is determined by the exponential distribution, with the transition rate as a parameter. CTMCs have a range of additional properties that can be analysed such as the average "sojourn time", the average time spent in a state before some transition occurs.

Finally, it is worth briefly mentioning Markov decision processes, these extend Markov chains to include non-determinism. The choice between transitions may be either non-deterministic or probabilistic. Like Markov chains they can be discrete time (DTMDPs) or continuous time (CTMDPs). Due to the non-determinism, the analysis of Markov decision processes involves finding lower or upper bounds to the properties of interest, otherwise the modelling and analysis is similar to that of Markov chains.

2.2 Formal Methods

The term *formal methods* encompasses a range of modelling techniques for analysing computer programs and systems. The key feature that these approaches have in common is a sound mathematical basis for determining whether the system being modelled respects some desirable properties [16]. An important aspect of software engineering is determining whether or not a design/model/implementation of a computer-based system behaves according to some "specification". The *specification* of a computer system is a document that records the desired behaviour of the system. Specifications may be *formal*, expressed in a notation that has a mathematically-defined semantics, or *informal*, written in natural language. Throughout this thesis the term specification will be used to describe the formal document unless otherwise stated.

There are many different formal methods, which roughly fall into the categories: finite state machines (also known as automata); Petri nets; process algebras (also known as process calculi); temporal logics; algebraic logics; model-based languages. Some of these favour a diagrammatic notation such as finite state machines and petri nets, whilst the others have a written notation. Some focus specifically on concurrent systems such as petri nets and process algebras, whilst others such as algebraic and model-based languages have more general applicability. Temporal logics focus on the timing aspects of the system. This thesis focuses on model-based specification languages, described in more detail in Section 2.2.1.

There are two types of *verification* techniques that are used to ensure that a model respects some desired property. These are: *model checking* "a technique that relies on building a finite model of a system and checking that a desired property holds in that model" [16]; and *theorem proving* "the process of finding a proof of a property from the axioms of the system" [16]. This thesis focuses on the latter. Many different logics are used in theorem proving, depending on the system assumptions and properties to be proved. Traditional logics such as boolean and predicate logic are common. More sophisticated logics include: three-valued logics to allow for undefinedness; temporal logics for timing properties; and of course probabilistic logics for analysing probabilistic systems.

An overview of the development of formal methods research is given by Jones [46]. The following section provides more details on the key features of model-based spec-

ification languages, before introducing the language upon which this thesis is based, Dijkstra's Guarded Command Language (GCL) [21].

2.2.1 Model-based specification languages

Recall (Chapter 1) that model-based specification languages use an abstract system model to communicate desired system behaviour (as opposed to listing properties as axioms). As a type of a formal specification language, these have a formal "semantics". The *semantics* of a language is a mathematical model that describes what each construct does and provides the basis for proving properties of models written in the language. Examples of model-based specification languages include: GCL [21]; B [2]; Event-B [3]; VDM [45, 27]; and Z [85]. In particular, this thesis is concerned with probabilistic extensions to GCL. An overview of GCL is given in Section 2.2.2 and McIver and Morgan's probabilistic extension to GCL is summarised in Section 2.3.2.

Two key features of specification languages such as those given above are "abstraction" and "refinement". *Abstraction* is the process of generalising programs or models with similar properties into a single model that captures the essential features of all of them. *Refinement* is the complementary process that adds implementation details to a model. The sorting algorithms quick sort, merge sort and bubble sort, for example, are all refinements of the more abstract process of sorting. The most abstract model of a program states *what* it should do, whereas the most refined model states *how* it should achieve it. The power of abstraction and refinement is that abstract models tend to be relatively small and simple, in which key properties of the program can be proven. It is then possible to prove that the more complex refinements of the abstract model respect the abstraction and thus also have the required properties of the program [44]. A number of refinement steps may be taken before the model is close to something that can be implemented in a programming language. Conversely, if the program is the starting point, a number of abstraction steps may be taken before the model is simple enough to permit proof of the required properties.

Various techniques can be used to provide abstraction in model-based specification languages. These include "implicit specification", abstract data types, and nondeterminism.

An *implicit specification* consists of pre- and post-condition pairs. A *pre-condition* is a predicate over the states of the model that is assumed to be true before a section of code is executed. A *post-condition* is a predicate that relates the initial and final states of a section of code. Such conditions can be used alongside the code to give extra information about the program. In this situation proof techniques can be used to show that the program respects the post-condition when the pre-condition is satisfied. However, pre- and post-conditions can also be used on their own as an abstraction mechanism to state *what* a section should do and when it can be executed, without giving the details on how it will be implemented. For example, the abstract specification of a square root

function would have a post-condition stating that the square of the result is equal to the input, and a pre-condition requiring the input to be non-negative.

Programming languages can have complicated data types that relate to how data is stored in memory. Model-based specification languages use *abstract data types* that approximate these, but are not restricted by how the data may be stored. For example, the abstract integer data type can take any whole number, whereas a programming language would be restricted to numbers that can represented in a certain number of bytes. Examples of abstract data types include: booleans; natural numbers; integers; real numbers; sets; sequences; maps; etc. These types can be further restricted where necessary through the use of "invariants". An *invariant* is a predicate over the state space that must always hold. Some languages, such as Event-B [3], do not have formal types and just use invariants to restrict the set of values a variable can take.

Non-determinism provides a way of recording that there may be more than one correct solution to a problem, without having to choose which answer will be provided by the program. For example, the square root function of a positive number always has two solutions – the positive root and the negative one. Non-determinism can be used to specify both alternatives without stating which should be used. However, nondeterminism can make proof difficult if the final value of a variable is unknown for some initial state. To resolve this issue two complementary interpretations of non-determinism have been defined: "angelic" and "demonic". For *angelic* non-determinism ($\sqcup$) it is assumed that the best possible option (w.r.t. the property being analysed) is always chosen. For *demonic* non-determinism ($\sqcap$) the opposite, that the worst option is chosen, is assumed.

2.2.2 Introducing GCL

Dijkstra created the Guarded Command Language [21] for the rigorous analysis and development of algorithms. Morgan and McIver chose GCL as the basis of their probabilistic language, pGCL, because it contains just the essential features, and no clutter [61]. The same reasoning is behind the choice of GCL as the base language for the research presented in this thesis.

Constructs in GCL take the general form of a guard followed by a command[1]. If the guard holds, the command may be executed. If multiple guards hold then any one of the commands whose guard holds may be executed. The selection of the command to execute in this case follows demonic non-determinism. The commands available in GCL include the usual: assignment; sequential composition; conditional choice; and while-loops. There are also two special commands, skip and abort. The skip command represents the program that does nothing, whilst the abort command represents the program that can do anything – this is usually used to represent non-termination.

Prior to GCL, the meanings of programs were invariably presented as Hoare triples

[1] Note that in the creation of pGCL, McIver and Morgan dropped the guards and included demonic choice explicitly.

[39] of the form

$$\{pre\}\ prog\ \{post\}\ ,$$

which means that from any initial state satisfying *pre* the program *prog* is guaranteed to terminate in a state satisfying *post*[2]. Dijkstra [21] presented an alternative way of writing this as[3]

$$pre \implies wp.prog.post\ ,$$

which has the same meaning, but the *wp* operator provides a more goal directed approach to specifying systems. The *weakest pre-condition* operator *wp* is defined such that for program *prog* and post-condition *post*, *wp.prog.post* returns the weakest pre-condition *p* that satisfies the Hoare triple above, i.e. any pre-condition *pre* satisfying the Hoare triple implies that *p* holds.

Dijkstra uses the *wp* operator to define the meaning of the commands found in GCL. As an example consider the assignment statement

$$x := E\ ,$$

where x is a program variable and E is an expression over the program variables of the appropriate type. The weakest pre-condition semantics of assignment is

$$wp.\left(x := E\right).post = post[x \backslash E]\ ,$$

the post-condition with all free variables of x replaced by the expression E. Consider the program

$$x := x + 1\ ,$$

and the post-condition $x > 5$. The weakest-precondition in this case is simply $x + 1 > 5$, i.e. $x > 4$. The weakest pre-condition semantics for the majority of the commands is summarised in Figure 2.4. However, it is very hard (or impossible) to determine the weakest pre-condition for an arbitrary loop, such that it is guaranteed to terminate [21, Chapter 6]. Therefore the usual approach to analysing loops is to find a "variant" and an "invariant". The *variant* is a finite numeric value that decreases each iteration (and cannot go below zero), thus ensuring termination. The *invariant* is a condition that must hold at the start of the loop and be preserved by the loop body. The invariant must also be such that the loop terminates in a state that satisfies the required post-condition.

[2]Note that this is a *total correctness* interpretation. For *partial correctness* termination is not required.

[3]Note that the dot notation is used throughout the material on GCL (and its probabilistic extensions) to mean function application, where $f.x$ means that function f is applied to argument x. This associates to the left such that $f.g.x$ is equivalent to $(f(g))(x)$.

	prog	*wp.prog.Q*
Abortion	abort	*false*
Identity	skip	Q
Assignment	$x := E$	$Q[x\backslash E]$
Composition	$prog_1 ; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$	$(G \implies wp.prog_1.Q) \wedge$
	else $prog_2$ fi	$(\neg G \implies wp.prog_2.Q)$

x is a program variable; E is an expression in the program variables; $prog_1$ and $prog_2$ are programs; G is a Boolean-valued expression in the program variables; and Q is a predicate.

Given an expression Q, the meaning of $Q[x\backslash E]$ is the expression Q in which free occurrences of x have been replaced by expression E.

Figure 2.4: Notation and weakest pre-condition semantics of GCL

2.3 Probability in Formal Methods

A number of formal methods have been developed that include an element of random behaviour in a model. There are two main approaches to consider: probabilistic branching and probabilistic assignment.

In *probabilistic branching* the formal method includes the option of a *probabilistic choice* between two or more alternative computations. In this thesis the notation ${}_p\oplus$ is used to represent this where $p \in [0, 1]$. For example, the program $prog_1 \ {}_p\oplus \ prog_2$ represents a system that behaves as $prog_1$ in $p * 100\%$ of the times it is executed and $prog_2$ in the rest. For more than two possible outcomes the probabilistic choice is written as $prog_1 \ @ \ p_1 \mid \cdots \mid prog_n \ @ \ p_n$ meaning that the program behaves as $prog_i$ with probability at least[4] p_i for each of the alternatives specified, where $\sum_i p_i \leq 1$. Note that the probabilities may sum to less than one to include the possibility of non-termination (with probability $1 - \sum_i p_i$).

In *probabilistic assignment* a value is assigned to a variable according to some specified probability distribution. Such probability distributions may be taken from a standard library of distributions or defined using the probability density/mass function (see Section 2.1.2). If the probability distribution is continuous, this may be referred to as *stochastic assignment* instead. Note that for discrete probability distributions, probabilistic assignment is a special case of probabilistic branching.

The term *probabilistic* will be applied to formal methods that represent discrete probability (including probabilistic branching). Those that represent continuous probability will be called *stochastic*. A more detailed discussion of probabilistic and stochastic formal methods is presented below.

[4]Note that the probability p_i is a minimum bound. The program can behave like any $prog_i$ (or something entirely different) for the remaining probability when the p_i's do not sum to 1.

2.3.1 Probabilistic formal methods

There are a wide variety of probabilistic formal methods. In general these extend existing formal methods with the probabilistic choice operator. For example, there exist: numerous probabilistic process algebras such as PCSP [69], TPCCS [34] and $ACP^{+}_{\pi,drt}$ [4]; a variety of probabilistic logics such as the probabilistic Hoare logic pL [18] and probabilistic temporal logic PCTL [35]; probabilistic model checkers such as PRISM [57]; and algebraic approaches for probabilistic systems [65]. A number of probabilistic model-based specification languages also exist. As probabilistic model-based specification languages are the focus of this thesis, these are discussed in more depth below.

One of the early developments in probabilistic model-based specification languages was the probabilistic PDL developed by Kozen [53, 54]. This introduced a semantics and logic for reasoning about probabilistic systems, in particular Kozen created a proof theory for probabilistic systems [54] that has similarities with Dijkstra's weakest pre-condition approach [21]. Jones [44] developed the semantic basis for probabilistic programs further by defining a probabilistic powerdomain to integrate probability into domain theory. However, neither of these approaches include non-determinism, which plays an important role in providing abstraction in models. McIver and Morgan [61] tackled the challenge of combining non-determinism and probabilistic choice and developed the probabilistic specification language pGCL (see Sections 2.3.2 and 5.4). This was then developed further by Morgan [67] and Hoang [38] to produce a semantics and a toolkit for a probabilistic version of the B [2] specification language.

Other probabilistic model-based specification languages have also been developed. These include: a probabilistic version of Z [84], which uses a similar approach to that adopted in pGCL; and the exploration of refinement in probabilistic action systems [80] [81]. Morgan, Hoang and Abrial [68] proposed a probabilistic version of Event-B [3], although this has not been developed in detail (apart from the use of "qualitative probability" reasoning to determine "almost-certain" termination of loops [33]). Tarasyuk et al. [79, 78] have also investigated adding probabilistic choice to Event-B and the translation of a suitable subset of it to PRISM for analysis.

Whilst there are many probabilistic formal methods, no such library of research exists for the formal specification and analysis of continuous probability, as shall be discussed in Section 2.3.3. Before that, the main probabilistic formal method that this thesis is concerned with, pGCL, is introduced in more detail.

2.3.2 Introducing pGCL

The probabilistic specification language pGCL [61, 6] is used throughout the thesis for modelling and analysing probabilistic programs, first in a case study to explore the use of probability in formal methods (Chapter 3), second as a basis for a specification language that supports reasoning about continuous probability distributions (Chapters 6, 7 and

	prog	*wp.prog.Q*
Abortion	abort	0
Identity	skip	Q
Assignment	$x := E$	$Q[x\backslash E]$
Composition	$prog_1 ; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$ else $prog_2$ fi	$[G]\times wp.prog_1.Q + [\neg G]\times wp.prog_2.Q$
Nondet. choice	$prog_1 \sqcap prog_2$	$wp.prog_1.Q \sqcap wp.prog_2.Q$
Probability	$prog_1 \ {}_p\!\oplus\ prog_2$	$p{*}wp.prog_1.Q + (1{-}p){*}wp.prog_2.Q$
While-loop	do $G \rightarrow body$ od	$(\mathcal{F}X \cdot [G]\times wp.body.X + [\neg G]\times Q)$

x is a program variable; E is an expression in the program variables; $prog_1$ and $prog_2$ are probabilistic programs; G is a Boolean-valued expression in the program variables; p is a constant probability in $[0, 1]$; and Q is an expectation.

Given an expression Q, the meaning of $Q[x\backslash E]$ is the expression Q in which free occurrences of x have been replaced by expression E. $\mathcal{F}$ is the least fixed point operator w.r.t the ordering $\leq$ between expectations.

Scalar multiplication $*$, multiplication $\times$, addition $+$, subtraction $-$, minimum, $\sqcap$, and the comparison (such as $\leq$ and $<$) between expectations are defined by the usual point-wise extension of these operators as they apply to the real numbers. Multiplication and scalar multiplication have the highest precedence, followed by addition, subtraction, minimum and finally the comparison operators. Operators of equal precedence are evaluated from the left.

$[\cdot]$ is the function that takes a Boolean expression *false* to 0 and *true* to 1. For $\{0, 1\}$ real-valued functions, operation $\leq$ means the same as implication over predicates, and $\times$ represents conjunction. Addition over disjoint predicates is equivalent to disjunction.

Figure 2.5: Notation and weakest pre-condition semantics of pGCL [6]

8). In this section the basics of pGCL are introduced in sufficient detail to understand the material presented in Chapter 3. A more detailed description of pGCL and its semantics is given in Section 5.4, knowledge of which is required to understand the material in Chapters 6, 7 and 8.

The language pGCL extends standard GCL (Section 2.2.2) to include probabilistic choice (see Section 2.3). Like GCL, it is a formalism that allows source-level reasoning about programs; it is a generalisation since it is able to handle *probabilistic* (as well as standard) properties [6].

Modelling in pGCL

The syntax of pGCL is given in Figure 2.5. In particular note the probabilistic choice operator (${}_p\!\oplus$) that allows the possibility of probabilistic updates. Thus $x := 1 \ {}_p\!\oplus\ x := 2$, would mean that x is assigned the value 1 with probability p, and 2 otherwise (with probability $1 - p$). With this, properties of interest are no longer necessarily absolute, but rather it is possible to reason about the probability that a property is established or to determine the *expected* value of a random variable of interest [6].

Analysis of pGCL models

A pGCL model can include annotations, real-valued expressions over the state space of the model, which from now on will be termed *expectations* as it is the *expected* value of these annotations that is of interest. For example,

$$\{p\} \quad x := 1 \ {}_p\!\oplus\ x := 2 \quad \{[x = 1]\} \tag{2.1}$$

$$\{p + 2(1-p)\} \quad x := 1 \ {}_p\!\oplus\ x := 2 \quad \{x\} \tag{2.2}$$

The post-expectation is treated as a random variable over the program variables. In (2.1), the notation $[x = 1]$ represents the characteristic random variable that returns 0 when "$x = 1$" is false and 1 when "$x = 1$" is true. In (2.2) the random variable is simply the value of x. The pre-expectation is the expected value of the random variable after execution of the program, thus for (2.1) it is p because the probabilistic update establishes "$x = 1$" with probability p; in (2.2) the pre-expectation is $p+2(1-p)$ because that is the expected value of x. More generally the pre-expectations will be sensitive to the initial state.

Formally an expectation is interpreted using the source-level *wp*-semantics set out in Figure 2.5, so that $\{P\}\ \ prog\ \ \{Q\}$ is valid provided that $P \leq wp.prog.Q$.

The idea conveniently generalises loop "invariants" as follows. Consider the while-loop in Figure 2.5. A standard loop invariant, I, has to hold at the start of every iteration, and constrains the states that the loop can enter, formally written as $G \wedge I \Rightarrow wp.body.I$. A *quantitative invariant*[5], E, can also be defined for such a loop. The expected value of E cannot decrease throughout the loop, written $[G] \times E \leq\ wp.body.E$.

Like standard invariants, quantitative invariants can be used to reason about properties of the whole iteration. For example, for standard invariant I of the above loop, if the *loop* terminates (with probability 1) then $I \Rightarrow wp.loop.I$. Similarly, $E \leq wp.loop.E$ holds for quantitative invariant E if *loop* is certainly terminating [6].

The use of quantitative invariants for reasoning about probabilistic loops is illustrated in the flash filestore case study (Chapter 3).

2.3.3 Stochastic formal methods

The number of stochastic formal methods is significantly smaller than the number of probabilistic formal methods. Further, the majority of stochastic formal methods are restricted to those systems that can be analysed by CTMCs (see Section 2.1.4). Examples of stochastic formal methods include: process algebras such as PEPA [37]; petri net approaches such as Generalised Stochastic Petri Nets [50] and Stochastic Activity Nets [74]; and also (semi-formal) architecture languages such as the error annex in AADL

[5] Quantitative invariants follow the same (*wp*) proof rules as expectations, but the term is considered more intuitive when considering loops.

[73]. As the above formalisms use CTMCs as their semantic basis, the continuous probability in them is limited to the exponential distribution. The stochastic model-based specification languages are not limited to the exponential distribution, but these are very scarce. They are discussed below along with some options for the verification of stochastic systems.

As with probabilistic model-based specification languages, Kozen [53, 54] pioneered the area of stochastic formal methods. Whilst the main focus of his research was on discrete probability, the language he developed (probabilistic PDL) supports continuous probability as well. His language contains no support for non-determinism or refinement, however. McIver et al. [63] also discuss the use of continuous probability when considering the partial correctness of loops. However, the discussion of continuous probability is essentially an aside and the details of a stochastic specification language are not worked out in full. For example, refinement is not considered and the interaction between continuous probability and non-determinism is not explored.

The interaction between theorem proving and stochastic behaviour has been explored by Hasan et al. [36]. They use the HOL theorem prover [32] to reason about expectation properties of continuous probability distributions. However, their research does not extend to the definition of a specification language that utilises their proof theory.

There also exist model checkers for the subset of stochastic languages that can be represented as CTMCs or CTMDPs (see Section 2.1.4). PRISM [57] provides support for model checking of CTMCs, whilst MRMC [51, 52] allows non-determinism as well (and thus supports CTMCs and CTMDPs). As with any approach that uses CTMCs for analysis, the continuous probability is limited to the exponential distribution. Further, whilst probabilistic model checkers are useful for some situations, state space explosion issues can make some systems expensive or impossible to analyse.

The above provides a promising start for the rigorous analysis of systems containing continuous probability. However, there are many gaps still to be resolved. In particular, this thesis focuses on: defining a stochastic model-based specification language that supports refinement; investigating the relationship between continuous probability and non-determinism; and exploring the practical application of a stochastic model-based specification language. As the closest language (other than pGCL) to that presented in this thesis, Kozen's probabilistic PDL is described in more detail below.

2.3.4 Probabilistic PDL

The language described by Kozen [54], probabilistic PDL, is similar in many ways to pGCL (McIver and Morgan used probabilistic PDL as a basis for pGCL). However, Kozen introduced probability as a replacement for non-determinism, and therefore (unlike pGCL) probabilistic PDL does not include any non-determinism. As they are based on two different (non-probabilistic) languages, the syntax of probabilistic PDL is quite different to pGCL. However, they both make use of an expectation transformer seman-

	prog	$< prog > f$
Positive linear comb.	$ap + bq$	$a < p > f + b < q > f$
Composition	$p; q$	$< p >< q > f$
Test	$B?$	Bf
Iteration	p^*	$f + < p; p^* > f$
Failure	fail	0
Identity	skip	f
Cond. choice	if B then p else q fi	$B < p > f + \neg B < q > f$

p and q are probabilistic programs; B is a Boolean-valued expression in the program variables; a, b are non-negative real numbers; * represents finite iteration; and f is a measurable function.

Note that the constructs in the second half of the table can be defined in terms of the constructs in the first half. Failure and identity can be defined as tests: fail $= 0?$, skip $= 1?$, and conditional choice can be defined in terms of tests and linear combination if B then p else q fi $=$ B?p $+ \neg$B?q.

A while-loop is defined in terms of iteration, test and composition as while B do $p = (B?p)^*; \neg B?$

Figure 2.6: Notation and expectation transformer semantics of probabilistic PDL

tics, for reasoning about probabilistic programs.

The syntax of probabilistic PDL and its expectation transformer semantics is summarised in Figure 2.6. The notation $< prog > f$ has a similar meaning to $wp.prog.f$ in pGCL. Notice that probabilistic choice is implicitly included as a positive linear combination, and that Kozen's failure construct is the same as abortion in pGCL. The syntax and transformer semantics of assignment (stochastic or otherwise) is not given in detail by Kozen. He only states that a primitive program may be a deterministic assignment such as $x := x + y$ or a random assignment $x := rnd$. As no examples are given that make use of random assignment, the meaning of it is not clear, but it is assumed that rnd can represent any probability distribution, discrete or continuous. Earlier versions of Kozen's research [53] discuss the use of a single probability distribution from which a random assignment is taken. It is unclear whether probabilistic PDL is intended to handle more than one probability distribution in any given program or not. There is nothing inherent in the semantics to prohibit multiple distributions, however.

Kozen's language makes a good starting point for rigorous reasoning about stochastic systems. However, this needs to be developed to clarify the meaning of the random assignment and its usage in practice. Probabilistic PDL also lacks support for non-determinism and refinement. These are important for abstraction and the step-wise development of systems using formal methods, respectively. The research presented in Parts 2 and 3 of this thesis aim to address this gap, although including non-determinism proves to be challenging and is not fully resolved. The rest of Part 1 explores the practical usage of probability in formal methods to inform the development of the new language in Part 2.

Chapter 3

Probability in Formal Methods in Practice

This chapter explores the practical application of probability in formal methods. A case study on flash memory illustrates how a probabilistic model-based specification language can be used to model and analyse probabilistic programs. This leads into a discussion of the kinds of questions that are more easily answered using continuous probability, first with respect to the case study, and then more generally. The case study is modelled in pGCL [61] as this forms the basis of the new stochastic language, described in Part 2 of this thesis. The discussion of continuous probability distributions motivates the stochastic extensions to pGCL.

The chapter begins with an overview of flash memory and why it makes a good case study (Section 3.1). A particularly interesting aspect of flash memory is then modelled and analysed in pGCL (Section 3.2). This is followed by a discussion of how the analysis could be extended with the use of continuous probability (Section 3.3). The discussion then moves away from the specifics of flash memory and examines more general application areas of continuous probability in computer-based systems (Section 3.4).

3.1 Flash Filestore Systems

A challenge problem in flash filestore management was proposed by Joshi et al. [48] as part of the Grand Challenge on verified software [47]. In response to this, there has been a wide variety of contributions from the formal methods community. For example: Butterfield and Woodcock [14] modelled and analysed the ONFI flash standard in Z; Kang and Jackson [49] modelled flash memory in Alloy; Damchoom et al. [19] investigated the data structure of the file system in Event-B; and Schierl et al. [75] developed a formal model of a real flash system (UBIFS) from its code. In this chapter, the first attempt to formalise the *probabilistic* aspects of flash filestores [6] is described.

Flash memory is a popular storage medium for many applications due to its lack

of moving parts. However, it also behaves differently to other storage media (such as magnetic disks) and new algorithms are required to deal with this new behaviour.

In particular, an individual bit stored on a flash memory cannot be overwritten; data has to be erased by block (between tens and hundreds of kilobytes) [28] before that space can be re-used. This is because individual bits can be cleared, but bits can be reliably set only a block at a time. To overcome this problem, data is not deleted immediately – pages (typically 512 bytes) are marked as one of the following:

- *valid* – contains data that is still in use by the system;
- *obsolete* – contains data that is marked for deletion (i.e. no longer in use);
- *clean* – does not contain any data.

When more space needs to be freed up, a *non-empty* block (i.e. a block with some valid and/or obsolete data on it) is selected. The valid data on the chosen block is moved to another location and the whole block is then erased (see Figure 3.1).

Another important feature of flash memory is that each block can only be erased a fixed number (typically 10,000 to 1,000,000) [28] of times before becoming unreliable.

The combination of the above characteristics leads to novel algorithms, designed to optimise the lifetime of the flash filestores [28]. These are known as "wear-levelling" algorithms, and are often probabilistic in nature.

3.1.1 Wear-levelling

As each block of flash memory has a limited number of times it can be erased, it is important to ensure that individual blocks do not become worn out prematurely. For example if the same block was used and erased repeatedly a significant number of times it would no longer operate reliably and the storage capacity of the flash filestore would have to be reduced. The process of ensuring that the relative number of erasures for each block in the filestore remains approximately the same at all times is called *wear-levelling*.

When designing and evaluating wear-levelling algorithms, there are (at least) two conflicting characteristics to consider: its impact on the lifetime of the flash memory; and the speed at which it frees up space. A naïve algorithm, that guarantees that no block has been erased more than (say) c times more than any other block, may yield a long lifetime (if c is small). However, it may also require a lot of relocation of data and therefore significantly affect the performance of the device. A number of probabilistic algorithms have been proposed for wear-levelling. It is hoped that these algorithms will result in the blocks being worn evenly on average, so that the expected lifetime of the system remains long, but that they will have better performance characteristics than their non-probabilistic counterparts.

In this case study, the probabilistic aspects of one particular probabilistic wear-levelling algorithm (taken from the JFFS flash file system [86]) are investigated. For

```
    GARBAGE COLLECTOR

*   do true →
*     select a non-empty block A;
      find another block B that has sufficient clean
        pages to receive the valid pages from A;
      copy them over from A to B;
*     erase A;
*   od
```

Figure 3.1: Abstract representation of a flash filestore garbage collector

99 of every 100 iterations this algorithm selects a block for erasure to maximise the amount of memory that will be freed up; for the remaining iteration a block is chosen at random for reclamation. The idea behind the JFFS algorithm is to choose the most efficient option (in terms of the space recovered) 99% of the time, but to also eliminate the possibility of certain blocks never being erased. For example, consider the situation where a large operating system file occupies a whole block. This file may never need to be changed over the lifetime of the filestore, and thus the block containing it would never be selected for erasure for efficiency reasons. However, the block has a chance of being selected by the random choice every 100 iterations, thus releasing a block that has not been erased as many times as other blocks.

3.2 Specification of a Flash Filestore

The previous section motivated the use of probabilistic algorithms in flash filestores. However, there has been little or no research into formally analysing the probabilistic aspects of these algorithms. In this section the probabilistic aspects of the JFFS wear-levelling algorithm are modelled and analysed in pGCL. The analysis determines the expected lifetime of a flash filestore that implements such an algorithm. For ease of analysis, the pGCL model of the JFFS algorithm is rather abstract. A discussion of how the model and analysis could be improved, in particular with the use of continuous probability, can be found in Section 3.3. The reader is referred to Section 2.3.2 for an introduction to pGCL that is sufficient to understand its use in this case study.

Before considering the JFFS wear-levelling algorithm specifically, the role of such an algorithm, in the bigger picture of a flash filestore, is (semi-formally) examined. In particular, the garbage collector functionality of a flash filestore is defined, as shown in Figure 3.1. It is assumed that the garbage collector is a continual loop that runs alongside functionality to read from, and write to, the filestore. In every iteration the garbage collector selects two blocks: one (A) to erase and one (B) to copy any valid data from A to.

The wear-levelling algorithm is the part of this process responsible for choosing the

$$\begin{array}{ll} \mathsf{do} & m < N \wedge n < N \rightarrow \\ & m := m + 1 \ {}_{\frac{1}{2}}\!\oplus\! \ n := n + 1; \\ & e := e + 1 \\ \mathsf{od} & \end{array}$$

- m represents the number of times block m has been erased, similarly for n
- e represents the total number of times any block has been erased
- N represents the maximum number of times a block can be erased

Figure 3.2: Specification of the probabilistic wear-levelling algorithm

blocks to be reclaimed. The relevant lines of the garbage collector have been marked with $*$ in Figure 3.1. In Section 3.2.1 these lines are expanded into a formal description of a simple probabilistic wear-levelling algorithm based on that used in the JFFS [86] flash file system.

3.2.1 Modelling a probabilistic wear-levelling algorithm in pGCL

In this section it is shown how to model a probabilistic wear-levelling algorithm using pGCL (see Section 2.3.2). The chosen algorithm is a simplified version of that used in JFFS [86]. One simplification is made to restrict the number of blocks in the filestore to two. A second is to only include the probabilistic iterations of the algorithm in the model, i.e. the iterations that select a block according to the amount of space reclaimed have been omitted. The simplifications are justified because the intention is to focus on the probabilistic aspects of the algorithm, and demonstrate how a probabilistic formalism can assist in the analysis of these aspects. The restriction on the number of blocks simplifies the arithmetic without losing the essence of the analysis.

The formal specification of the wear-levelling algorithm is shown in Figure 3.2. The variables m and n represent two different blocks, and record the number of times that each has been erased so far. In each iteration of the algorithm, one of these blocks is selected for reclamation: block m or block n, each with probability $\frac{1}{2}$. A further variable e keeps track of the total $m + n$. This is used in the analysis below to calculate the expected lifetime of the flash filestore in terms of number of erasures. It is assumed that if either of the blocks reach some maximum number of erasures, N, then the flash filestore is retired.

3.2.2 Analysis

The probabilistic wear-levelling algorithm makes no guarantee that each block has been erased at most c (for $c < N$) times more than any other block. However, the quantitative invariant (Section 2.3.2)

$$n - m$$

can be used to show that the average difference between the number of erasures of each block is zero (after each execution of the body of the loop). Intuitively, this property holds because of the symmetry of the probabilistic choice.

But what impact does achieving wear-levelling have on the lifetime of the flash filestore? The wear-levelling algorithm above is now analysed to determine the expected lifetime it provides. The lifetime of the device was chosen to be measured in terms of the number of erasures. This involved determining the expected value of the (random) variable e on termination of the loop, written $E[e]$, which is calculated with the assistance of loop "invariants" (Section 2.3.2).

Using the standard loop invariant

$$e = m + n$$

gives $E[e] = E[m + n]$, and so it is enough to calculate the expected value of $m + n$; but to do this a complex quantitative invariant is required. Knowledge of the negative binomial probability distribution provides one.

Recall (Section 2.1.2 or [71]) that the negative binomial distribution models the number of trials required until x instances of a specific event have been observed, assuming that the probability of the event occurring is constant across all trials. Consider the expected value of m alone initially: an increment in m can be thought of as the event of interest for a negative binomial distribution, with N being the target number of increments required. However, this distribution allows situations not permitted by our model, for example, the case where $m = N$ and $n = N + 1$. It is also necessary to include the situation where n reaches N before m does. To resolve this the negative binomial distribution is adapted to have an upper bound on the number of trials allowed, and to have two instances of the distribution – one each for the situations in which m and n reach N first. Using the basis of the negative binomial distribution, and incorporating the complications described above, it turns out that there exists another quantitative invariant of the loop, as shown in Formula 3.1. Intuitively this equation assumes that $m + n$ trials have already occurred (first line) and calculates the expected number of remaining trials given this fact (second line).

$$\begin{array}{c} m + n + \\ \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e + \displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \tag{3.1}$$

It can be formally confirmed that the expression (call it inv) shown in Formula 3.1 is a quantitative invariant of the loop. This requires (Section 2.3.2) showing that $[G] \times inv \leq wp.body.inv$. Using the wp proof rules defined in Figure 2.5 it can be shown that

$wp.body.inv \equiv inv$, as is summarised below (a more detailed version of this proof can be found in Appendix A). It can also be trivially proved (see Appendix A) that the values of m and n on termination are as expected (e.g. that $inv\,(N, n, e) = N + n$).

$$
\begin{array}{lll}
 & wp.body.inv & \\
\equiv & wp.\left(\left(m := m+1 \;{}_{\frac{1}{2}}\!\oplus\; n := n+1\right);\; e := e+1\right).inv & \text{definition of } body \\
\equiv & & \text{composition, assignment, probability and definition of } inv \\
 & \frac{1}{2} * \left(\begin{array}{l} m+1+n+ \displaystyle\sum_{e=N-(m+1)}^{2N-((m+1)+n+1)} e \left(\begin{array}{c} e-1 \\ N-(m+1)-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-n}^{2N-((m+1)+n+1)} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \right) & \\
 & + \quad \frac{1}{2} * \left(\begin{array}{l} m+n+1+ \displaystyle\sum_{e=N-m}^{2N-(m+(n+1)+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-(n+1)}^{2N-(m+(n+1)+1)} e \left(\begin{array}{c} e-1 \\ N-(n+1)-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \right) & \\
\equiv & & \text{algebra including combination and summation rules} \\
 & m+n \quad + \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e & \\
 & \qquad + \displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e & \\
\equiv & inv & \text{definition of } inv
\end{array}
$$

The expected lifetime of the flash filestore ($E[e]$) can be determined from inv by substituting m and n with their initial values (both zero). This gives the expression found in Formula 3.2.

$$E[e] = 2 * \sum_{e=N}^{2N-1} e \left(\begin{array}{c} e-1 \\ N-1 \end{array} \right) \left(\frac{1}{2}\right)^e \tag{3.2}$$

Using the expression in Formula 3.2, the expected lifetime of the flash filestore was calculated for various values of N, tabulated in Figure 3.3. It can be seen that $E[e]/N$ approaches two as N gets larger (Figure 3.3). Note that an algorithm that always alternated between blocks (i.e. ensured each block had been erased at most once more than the other) would have an expected lifetime of $2N - 1$, for any N. In spite of this, the JFFS wear-levelling algorithm advocates the use of a random choice, perhaps because this does not require any historical data about block erasures. The justification (for using a random choice of blocks) is irrelevant, however, for demonstrating the use of probabilistic formal methods via the analysis of the probabilistic aspects of the JFFS

N	E[e]	E[e]/N
2	2.50	1.25
3	4.13	1.38
4	5.81	1.45
5	7.54	1.51
10	16.48	1.65
50	92.04	1.84
100	188.73	1.89
500	974.77	1.95
1000	1964.32	1.96

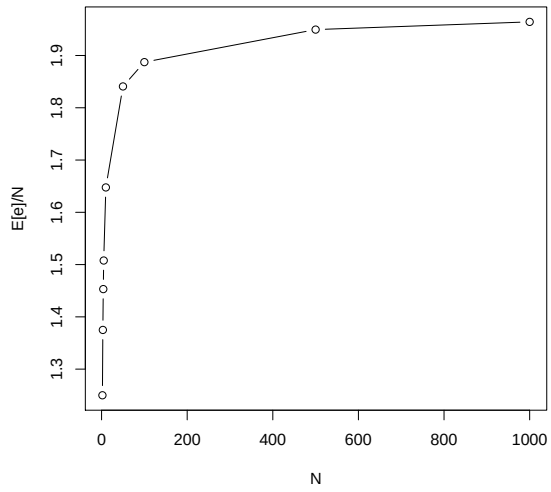

Figure 3.3: Expected lifetime ($E[e]$) for the probabilistic wear-levelling algorithm

wear-levelling algorithm in pGCL.

3.3 Continuous Probability in Flash Filestores

The previous section demonstrated the use of a model-based specification language for modelling and analysing the probabilistic aspects of a flash filestore wear-levelling algorithm. The model abstracts away from a lot of the details of a real system, however, which means that a number of interesting questions have to be neglected in its analysis. This section discusses some of the questions that deserve further exploration, but cannot be answered by the model presented. Questions that are more easily answered with the use of continuous probability are of particular interest. The discussion focuses on two dependability related problems: what it means for a flash filestore to fail and require replacement; and the need for performance-reliability trade-offs. Throughout the discussion, the role of continuous probability distributions is highlighted.

3.3.1 When does a flash filestore fail?

Deciding when a flash filestore is no longer useful or reliable (and needs to be replaced with another) is not always straightforward. In particular, if the disk in question is to be used in a remote location (in outer space in the extreme case), the cost associated with replacement may be high. In other applications the filestore may be straightforward and cheap to replace. In the former situation, it would be necessary to estimate the lifetime of the disk before its deployment. A number of different strategies for determining when a disk should be replaced are discussed below, along with the analysis that would be required to determine their expected lifetime.

A block reaches its maximum number of erasures

The simplest retirement strategy to model and analyse is that considered in Section 3.2. This strategy chose to retire the flash filestore as soon as a single block had reached its (recommended) maximum number of erasures. Beyond this number of erasures, the reliability of the block is thought to drop below acceptable levels. However, even if one block can no longer be used, there may be a large proportion of the filestore that is still serviceable. Therefore, for most situations this approach seems a little extreme, but may be necessary for applications that require a high percentage of the total disk space to operate correctly.

For some situations, it is sufficient to analyse the expected lifetime of a flash filestore that implements this retirement strategy (without continuous probability) as shown above. However, a more accurate analysis would need to take into account the usage of the flash filestore (in terms of the size and rate of write and deletion operations), to determine the behaviour of the non-probabilistic iterations of the JFFS algorithm. In some situations all of this data may be available and the wear-levelling algorithm may be modelled precisely. However, it is more likely that limited usage data is available and that the rate of write operations, for example, may only be approximated by a continuous probability distribution such as the exponential distribution. In this situation the use of continuous probability provides an indication of the expected lifetime of the flash filestore in the absence of complete information about how it will be used.

A given proportion of blocks reach the maximum number of erasures

A simple extension to the above strategy is to retire the flash filestore when a given proportion of blocks reach their maximum number of erasures. The choice of this proportion would depend on how much disk space is required for the installed applications to operate correctly. Individual blocks would be retired once they reached the maximum number of erasures, i.e. these would be marked as unusable, and the flash filestore would proceed to operate as if the retired blocks did not exist.

The analysis approach for this strategy would be similar to that described above, although the complexity of the analysis would increase as the number of blocks to choose from would not remain consistent throughout the lifetime of the flash filestore.

Valid data cannot be relocated

More complex strategies use the approach of allowing individual blocks to be retired as described above. The criterion for retiring the whole filestore, however, are not based on the number of blocks reaching their maximum number of erasures. The first of the more complex strategies is to retire the flash filestore when an erase procedure is not possible due to insufficient free space on the filestore. When a block is erased, it is necessary to find sufficient free space to move the valid data remaining on the chosen block to. If

sufficient space cannot be found regardless of the block chosen for erasure, the filestore should be retired immediately otherwise data will be lost.

To model this situation it is necessary to know the number of valid, obsolete and clean pages on each block every time the wear-levelling algorithm chooses a block to erase. This could be modelled using a (discrete) multinomial distribution [71] that states the probability of each page in each block being either valid, obsolete or clean. However, a realistic model of the system requires data about the usage of the flash filestore as before. Again, it is likely that this information will need to be modelled using continuous probability distributions such as the exponential distribution.

Data cannot be written

The second of the more complex strategies is to retire the flash filestore if a write operation is requested and insufficient free space exists for the data to be written. Whilst write operations are not handled by the garbage collector modelled above, this is still an important situation to consider. This strategy may be relevant in situations where the incoming data rate is high and there is a limited buffer in which data pending a write operation can be stored. Once more, it is assumed that individual blocks are retired when they reach their maximum number of erasures.

In order to model this strategy, data would be needed on the rate at which write requests occur (including the size of the requests) and the rate at which write requests are processed. Continuous probability distributions may be used to model: the time between write operations; the size of the write operations; and the rate at which data is processed from the write buffer of the flash filestore.

Determining the maximum number of erasures

The final point to discuss is not really a retirement strategy, but an exploration of how a suitable maximum number of erasures per block may be determined. This requires a model of the failure rate of the individual bits of memory. Flash memory has been analysed by electrical engineers and the time to failure of a bit in flash memory found to occur according to a probability distribution called the Fréchet distribution [40]. This could be used to determine a suitable maximum number of erasures to set for a block, based upon an acceptable level of risk of data loss. The Fréchet distribution is a continuous probability distribution, therefore this analysis clearly requires the use of continuous probability.

3.3.2 Considering trade-offs

In Section 3.1.1 an example was given where the choice of wear-levelling algorithm affected the performance of a flash filestore. If sufficient data were included in the model of the flash filestore and wear-levelling algorithm about the read, write and deletion rates

of data it may also be possible to analyse the trade-off between the expected lifetime of the flash filestore and the performance issues. Such analysis is complex and would benefit from the formal modelling of continuous probability distributions.

It is conceivable that other trade-offs may also be of interest. For example, the size (cost) of the write buffer (that stores incoming data until it is written to the flash filestore) may be traded off against the probability of losing data. Many of these trade-offs would either require (or have simpler reasoning with) the use of continuous probability distributions.

3.4 Applications of Continuous Probability in Computing

In this section the wider application of continuous probability in computing is briefly discussed. There are many application areas of continuous probability. This section only intends to provide a sample of these. In particular, the focus is on the use of continuous probability distributions for dependability analysis.

The area of embedded systems (safety-critical or otherwise) possibly has the most prevalent occurrence of continuous probability in computing. This is because the behaviour of an embedded system cannot be considered in isolation to the hardware that it controls. Such hardware has (moving) parts that deteriorate over time. This deterioration process is typically modelled by a continuous probability distribution that provides the probability of the hardware failing over time. Sometimes this failure distribution is given by a single rate parameter that relates to the exponential distribution. However, more complex (continuous) probability distributions are often more appropriate. For example, the "bathtub curve" models components that are more likely to fail early or late in their lifetime than during the middle of their lifetime. Another example of a continuous probability used to model component failures is the Fréchet distribution discussed above for the failure of bits of flash memory. Failure distributions may also be applied to software failures, but this is much rarer as software is generally not considered to fail by chance.

Another example of where continuous probability occurs in embedded systems is in (analogue) sensor reading errors. Analogue sensors are typically not 100% accurate, i.e. the reading generally differs from the actual value it is intended to measure. The difference between the actual value and the reading (the sensor error) can be modelled by a continuous probability distribution. In the simplest situation, a (continuous) uniform distribution can be used to model the error, but more complex distributions (such as the normal distribution) may also be used. This application of continuous probability in computer-based systems is explored in more detail in Chapter 8.

Hasan et al. [36] present another application area for the continuous uniform distribution. They use it to model the round-off error in floating point arithmetic. This clearly applies to any computer-based system that relies on (accurate) floating point

arithmetic.

Performance analysis is another area where continuous probability is frequently used in computing [37]. The arrival rate and processing time of jobs are often modelled by the exponential distribution, to calculate performance metrics of a system such as its throughput. The hyperexponential distribution may be used instead in the situation where the system has a number of different modes. Whilst performance is not an attribute of dependability in itself, it often needs to be considered for dependable systems as dependability and performance tend to be conflicting concerns.

Continuous probability may be used in a variety of other situations (beyond those that are dependability related) in computing. For example: bio-technology applications may use continuous probability to model (random) biological processes; or scheduling algorithms may use continuous probability to model the processing time jobs require. These are not discussed in detail because this thesis is particularly concerned with modelling and analysing dependability.

It is worth noting at this point that due to necessary restrictions to the state space of sGCL models (see Chapter 6), the flash filestore case study is not the most suitable case study for revisiting with continuous probability.

3.5 Concluding Remarks

This chapter has provided a demonstration of how to apply a probabilistic model-based specification language to the analysis of dependability properties of computer-based systems. It also described the first attempt at formally analysing the probabilistic aspects of flash filestores. However, the analysis of flash filestores is limited when using a modelling language that only allows discrete probability. The extensions (to the analysis of flash filestores) that would be possible with the use of continuous probability, in addition to discrete probability, were discussed at length. This led into a wider discussion of the use of continuous probability in computing, particularly when modelling and analysing dependability attributes.

Chapter 4

Towards a Stochastic Model-Based Specification Language

The previous chapter demonstrated the use of probability in formal methods in practice and discussed the need for continuous probability in particular. The first half of this chapter illustrates how existing formalisms can be used to model and analyse a case study (on an emergency braking system of a train) that is stochastic in nature. For this discussion, two existing and complementary formalisms are chosen to model and analyse the case study. The first of these (PRISM) supports continuous probability, but is a model-checker (which has limited support for abstraction), not a model-based specification language. The second (pB) is a probabilistic model-based specification language, but does not support continuous probability. These formalisms were selected for their complementary features and accessibility. The intention is to provide a practical illustration of the strengths and limitations of formalisms discussed in Section 2.3.

The second half of this chapter develops a novel prototype stochastic model-based specification language (Stochastic Event-B [5]), and illustrates how this could be used to model and analyse the case study. Stochastic Event-B combines the use of continuous probability illustrated in PRISM, and the features of a model-based specification language found in pB. The use of Stochastic Event-B in analysing the case study is discussed, before evaluating the required simplifications to such a language for a full formal treatment (as presented in Part 2 of this thesis). In particular, it is beneficial to be able to remove non-determinism from the language because the semantic definition of a model-based specification language containing non-determinism and continuous probability is especially challenging.

The chapter begins with an overview of the emergency brake case study (Section 4.1). This is followed by the application of PRISM (Section 4.2) and pB (Section 4.3) to the case study. Stochastic Event-B is then introduced and also applied to the case study

(Section 4.4). The chapter concludes with a discussion of the (un)suitability of Stochastic Event-B for a full formal description (Section 4.5).

4.1 The Emergency Brake

This section describes the emergency brake case study that is used throughout this chapter. The purpose of the case study is to illustrate the strengths and limitations of existing formalisms, and to demonstrate the ideas behind Stochastic Event-B. Therefore the scenario considered is rather simple, and only loosely based on a real system.

The emergency brake case study is derived from an industrial problem considered in the DEPLOY project[1] [26]. A train has an emergency brake (EB) that can either be *applied* (the brake is on and the train is stopping) or *not applied* (the brake is off and has no effect on the speed of the train). Some external system (either a person or another computer system) can *command* the brake to be *applied* at any time.

The emergency brake system can fail in two possible ways:

- An *unsafe failure* occurs when the emergency brake has been *commanded*, but not *applied*;
- A *safe failure* occurs when the emergency brake is *applied*, even though it has not been *commanded*.

For simplicity, recovery from emergency brake requests and failures is ignored in this study.

There are two types of events that can occur in this system. Those that occur in time according to some average transition rate and those that occur instantaneously, triggered by some other state change. An example of the former is the rate at which the emergency brake is requested – events like this are most naturally modelled by the exponential distribution. An example of an instantaneous transition is whether, after an emergency brake request has occurred, the emergency brake is applied or has an unsafe failure. This type of event is most naturally modelled as a probabilistic choice between the two options, occurring immediately after the state change triggering the choice. Therefore, an intuitive model of this system should include state transitions according to the exponential distribution as well as instantaneous state updates with probabilistic choice.

Figure 4.1 shows the transitions that can occur in the emergency brake system considered. State 1 is the initial state in which the EB is not applied and has not been commanded. From the initial state there are two possibilities. An emergency brake request can occur taking the system to state 2 where $EB\_command$ is set – this is modelled according to the exponential distribution with rate λ_{req}. State 2 is considered

[1] The EU FP7 project on industrial deployment of advanced system engineering methods for high productivity and dependability (Grant Agreement Number: 214158), www.deploy-project.eu.

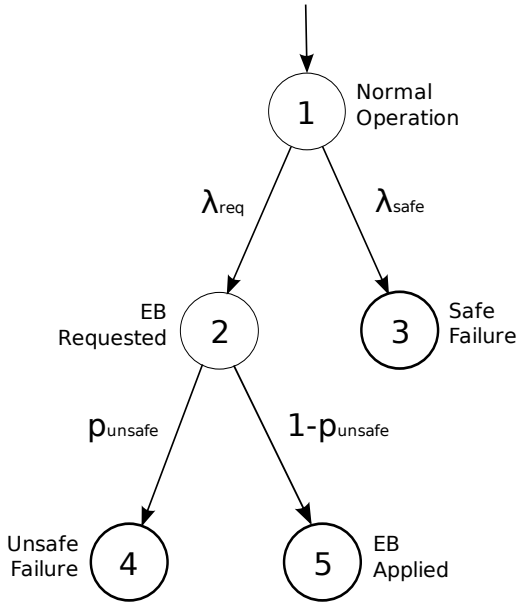

Figure 4.1: States and transitions for the EB system

to be a transient state from which one of two options will occur instantaneously. The first possible transition from state 2 is that of an unsafe failure (state 4), where the emergency brake is not set – this occurs with some probability, p_{unsafe}. The other transition from state 2 is a normal application of the emergency brake (state 5), i.e. *EB_applied* is set – this occurs with probability $1 - p_{unsafe}$. The final transition that should be mentioned is the safe failure transition, which occurs from state 1 and takes the system to the safe failure state (state 3) in which *EB_applied* is set, but *EB_command* is not. This transition is considered to occur according to the exponential distribution with rate λ_{safe}.

There is a safety objective [26] that "$unsafe\ situation \leq \lambda_{max}/hour$". For the purposes of this case study, the safety objective is interpreted to mean that (on average) less than λ_{max} *transitions* into an unsafe situation occur per hour, where an unsafe situation is represented by an unsafe failure.

4.2 Modelling the Emergency Brake in PRISM

This section introduces the probabilistic model checker PRISM [57], and applies it to the case study. The strengths and limitations of PRISM, as illustrated by the case study, are then discussed. The feature of PRISM that is of particular interest is its ability to model and analyse continuous probability (in terms of the exponential distribution). It also has excellent tool support that is readily available. It is not, however, a model-based specification language, which means that its support for abstraction is limited.

4.2.1 PRISM overview

PRISM is a probabilistic model checker that provides support for three different types of probabilistic models: Discrete Time Markov Chains (DTMCs); Continuous Time Markov Chains (CTMCs); and Discrete Time Markov Decision Processes (DTMDPs). Of these, the CTMC model is of interest as this supports the exponential distribution.

The behaviour of a CTMC model in PRISM is given through a set of guarded commands that model the state transitions, and the conditions under which they occur. Each state transition has an associated rate, which indicates the timing aspects of the transition. The rate determines how much time passes before an enabled transition occurs (according to the exponential distribution). A summary of the syntax of the CTMC models in PRISM[2] is provided in Figure 4.2. Analysis in PRISM is achieved by allocating rewards to states and state transitions, and computing the expected value of such rewards (Kartson et al. give some examples of their use [50]). The syntax for rewards is also included in Figure 4.2.

4.2.2 PRISM models and analysis

Two PRISM models of the emergency brake case study were developed. The first consisted of just one module, in which all the behaviour was modelled. Figure 4.3 shows the unsafe failure event, the full model can be found in Appendix B.1.1. Realistically the subsystem for requesting the EB would be a separate subsystem to that responsible for applying the EB. Therefore the second PRISM model described these behaviours in separate modules. The *request* event (see Figure 4.4) is synchronised over both the modules to link their behaviour. The full model can be found in Appendix B.1.2. Both versions of the model are semantically equivalent. As PRISM only allows numerical (and not algebraic) analysis, the transition rates were given sample values for this purpose.

A safety analysis was carried out on both of the PRISM models described above. A reward of 1 was allocated to the unsafe failure state ($EB\_command \wedge \neg EB\_applied$), with a reward of 0 being allocated elsewhere. This reward is cumulative, i.e. for every time unit in which the model is in a unsafe state the reward is incremented by one. PRISM was then used to analyse the expected value of the reward (R) in steady state (S) by verifying the property $R =?[S]$. The result of this analysis can be interpreted as the probability that the final state is unsafe. Some example results are tabulated in Figure 4.5, these show that p_{unsafe} is a key parameter. Note that (as recovery is being ignored) there is no way of returning to the initial state once it has been left. This means that the steady state analysis described above provides no information about the amount of time spent in the initial state (the probability of staying in the initial state forever is negligible). It is also possible to analyse the *sojourn time*[3] in PRISM.

[2]For more details (and for information about modelling DTMCs and DTMDPs in PRISM) see http://www.prismmodelchecker.org/.

[3]Defined as the average time spent in some state until a transition occurs [50].

A PRISM model consists of a set of constant declarations, one or more modules, and a set of rewards. Each module consists of a number of variable declarations and a number of commands detailing the permitted state transitions. The syntax for constant, variable and command declarations is tabulated below.

Declaration	Syntax
Constant	**const** type name = value;
Variable	name : type **init** init_value;
Command	[action] guard -> rate : update;

A command is only *enabled* if the expression guard evaluates to true. The rate determines (stochastically according to the exponential distribution) how much time passes before an *enabled* event actually occurs. When an event occurs, the state transitions in update are implemented. Note that the rate clause is optional. If it is left blank there must be another command in a separate module with the same action with which this command is *synchronised*, i.e. when a command occurs in a module, all commands with the same action label in other modules are also triggered.

Rewards are used to provide a range of analysis in PRISM, they have the following syntax:

rewards "label"
  [action] guard : reward;
  [action2] guard2 : reward2;
  ...
endrewards

The occurrence of a transition labelled action from an originating state satisfying the guard expression is awarded the stated reward. The action clause is optional – if it is omitted, reward is acquired by being in a state that satisfies the guard expression.

Figure 4.2: PRISM syntax for CTMC models

```
[unsafe_failure]   commanded = false & applied = false ->
                     request_rate * unsafe_fail_prob : commanded' = true;
```

Figure 4.3: First PRISM model - unsafe failure event

```
module      EB_request
  ...
  [request]   commanded = false & applied = false ->
                request_rate : commanded' = true;
  ...
endmodule

module      EB_application
  ...
  [request]   true -> 1 - unsafe_fail_prob : applied' = true +
                unsafe_fail_prob : applied' = false;
  ...
endmodule
```

Figure 4.4: Second PRISM model - synchronised request event

However, for this case study such analysis would only reveal the average time before *any* of the three possible events occur and thus gives little insight into the *safety* of the system.

4.2.3 Experiences with PRISM

PRISM is a well-designed tool with a clear user interface and simple modelling language. This makes it particularly attractive to software engineers who may have little or no knowledge about statistical modelling. PRISM also has constructs that allow the user to write their model in synchronised modules, providing good structuring facilities for large or complex modules. Having a different section of the model for rewards annotations allows good separation of the information needed to model the functionality from that needed for analysing desired properties of the model. The ability to model *Continuous* Time Markov Chains (CTMCs) as well as discrete ones is a key advantage for modelling scenarios such as the one described in Section 4.1. Without these continuous variables, abstractions have to be made either at the cost of the detail in the model or the ease of modelling (see Section 4.3.2 for further discussion about this). However, because Markov Chains rely on the memoryless property (historical behaviour does not affect the probability of future behaviour), the only continuous distribution that can be modelled is the exponential distribution (which is not applicable in many situations).

A further limitation of PRISM is the lack of support for abstraction. There are no abstract data types, such as the natural numbers. Every parameter needs to be initialised and the range of values each variable can take needs to be recorded. The values

λ_{req}	λ_{safe}	p_{unsafe}	$R =?[S]$
0.1	0.001	0.01	0.00990
0.5	0.001	0.01	0.00998
0.1	0.005	0.01	0.00952
0.1	0.001	0.05	0.04955

Figure 4.5: Sample results of the emergency brake analysis in PRISM

of parameters are not always known at the early stages of design, instead it can be helpful to perform some algebraic analysis over models to determine the relationship between different parameters of the model and design requirements. Such algebraic analysis is not possible in PRISM. There is also no way to combine continuous probability and non-determinism in PRISM (although this is possible in other model checkers such as MRMC [52] as discussed in Section 2.3.3). With limited support for abstraction, there is also no notion of refinement in PRISM. It is not possible to determine equivalence or an ordering of PRISM models. This is likely to be due to it not being obvious how to define an ordering or equivalence for CTMCs in general, although there has been some research to address this for process algebras [37]. A final interesting feature of PRISM is that there is no real notion of termination. If the user wishes to model a terminating state, an infinite loop back to that final state is required.

Overall PRISM is a useful tool for analysing stochastic behaviour that can be captured by the exponential distribution. However, it is perhaps more useful in the later stages of the software lifecycle when most parameters of the model will have known values and no refinement of the model is required.

4.3 Modelling the Emergency Brake in pB

This section introduces the probabilistic model-based specification language pB [38], and applies it to the case study. The strengths and limitations of pB, as illustrated by the case study, are then discussed. The feature of interest in pB is its support for abstraction and refinement. It is also supported by a toolkit (although this is not cheap to obtain and as such was not used for this case study). The drawback of pB, however, is that it only supports discrete probability. This means that the stochastic behaviour inherent in the case study has to be approximated in some way for its analysis.

4.3.1 pB overview

Probabilistic B is essentially standard classical B [2] extended to include the probabilistic choice operator ($_p\oplus$), e.g. $x := 1 \ _p\oplus \ x := 2$. A pB model includes a number of variable declarations and operations over those variables. Each operation has a pre-condition to record the valid states from which the operation may be called. A novel construct,

	S	$[S]\ Q$
Assignment	$x := E$	$Q[x\backslash E]$
Pre-condition	$P \mid S$	$P \times [S]\ Q$
Choice	$S_1\ []\ S_2$	$[S_1]\ Q \min\ [S_2]\ Q$
Probability	$S_1\ {}_p\!\oplus\ S_2$	$p \times [S_1]\ Q + (1-p) \times [S_2]\ Q$
Composition	$S_1; S_2$	$[S_1]\ ([S_2]\ Q)$
Cond. choice	$P \implies S$	$1/P \times [S]\ Q$
Identity	skip	Q
Parameterised subs.	$@x \cdot S$	$\sqcap x \cdot ([S]\ Q)$

x is a variable (or vector of variables); E is an expression in the program variables; P is a predicate; S, S_1 and S_2 are probabilistic generalised substitutions; p is a constant probability in $[0, 1]$; and Q is an expectation. The expression $\sqcap x \cdot (E)$ is interpreted as the minimum of E for all x.

Given an expression Q, the meaning of $Q[x\backslash E]$ is the expression Q in which free occurrences of x have been replaced by expression E.

Figure 4.6: Notation and expectation transformer semantics of pGSL

known as an *expectation*, is used to define probabilistic properties that hold throughout the model. These are essentially probabilistic versions of invariants. The language shares similarities with pGCL (Section 2.3.2), in particular in the use of an expectation-transformer semantics for analysing properties of interest.

The syntax and semantics of pB are based on the probabilistic generalised substitution language (pGSL) [67]. This is given in terms of an expectation transformer semantics (similar to pGCL[4]), as shown in Figure 4.6. The interpretation of pGSL is (the same as pGCL – Section 2.3.2) that $[S]\ Q$ determines the expected value of Q after the substitution S has occurred. For composite substitutions, this is given in terms of the expected value of its constituent substitutions.

The *expectations* construct is used to analyse safety properties in the pB models of the emergency brake system. Therefore this will now be described in more detail. An *expectation* is a numeric expression over the (random) variables of the model. It is written $e \leq Inv$, where e is a value and Inv is an expression over the variables of the model. For a pB model to satisfy an expectation, it is required that the expected value of expression Inv is never below the value e (throughout the model). For example, consider a system that observes the number of heads (h) and tails occurring when a tossing a fair coin n times. An expectation of the system may be $0 \leq h - \frac{n}{2}$, meaning that heads account for *at least* half of the total number of observations. Note that a similar expectation about the number of tails observed would also be required to ensure that the coin is *fair*. The use of expectations leads to the following proof obligations: $e \leq [Init]Inv$ for initialisation $Init$; and $Inv \leq [Op]Inv$ for every operation (Op) in the pB model. Note that these are the probabilistic versions of invariant proof obligations.

[4]The $[S]\ Q$ notation has a similar meaning to *wp.prog.Q* in pGCL.

4.3.2 Discrete approximations of the emergency brake system

The pB formalism [38] has no representation of time or continuous probability distributions. Therefore to model the EB scenario in pB, first it is important to decide how best to abstract away from time and transform the stochastic behaviour into discrete probabilistic choice statements.

There are essentially two ways in which the probabilities can be calculated based on the rate (CTMC) based description of the system and the choice between them depends on what kind of questions are to be asked of the model. One question a system designer may like to consider is which state transition is likely to occur first from a given state – this scenario is described in Option 1. Alternatively, a designer may like to know what could happen in the next time step (including the possibility of nothing), or for how long the system is likely to remain in a state – this scenario is described in Option 2. The limitations and benefits of both of these options are discussed in more detail below.

Option 1: Embedded Markov Chain (What happens next?)

In this option the relative likelihood of each state transition occurring next (from the current state) is determined. This is a simple and popular way of abstracting a CTMC into a DTMC, known as the *Embedded Markov Chain*. The rates are converted into probabilities as follows:

$$p_{ij} = \begin{cases} \dfrac{\lambda_{ij}}{\sum_{k \neq i} \lambda_{ik}} & i \neq j \\ 0 & otherwise \end{cases} . \tag{4.1}$$

Recall (Section 2.1.4) that p_{ij} is the probability that a transition from state i to state j occurs, and λ_{ij} is the rate at which transitions from state i to state j occur in the CTMC. Applying the above to all possible transitions results in a DTMC that can then be modelled with probabilistic choice statements in pB.

Whilst this is a straightforward approach for calculating probabilities for use with pB, it is not without consequences. With a model like this all sense of how long is spent in each state is lost, and in fact for the EB scenario all that can be determined from this model is the probability of the system ending up in each of the terminating states. When interested in dependability data such as the mean time between failures this abstraction is clearly not very useful.

Option 2: Single time unit transitions (What happens in the next time unit?)

In the second option the question of interest is: what could happen within some small amount of time, t? This includes the possibility of nothing happening. The probability of each possible transition occurring within time t needs to be calculated. The prob-

ability of doing nothing within time t also needs to be calculated (by subtracting the probabilities of all the other possible transitions from one). In general, these calculations are far from straightforward, especially for large, complex systems. The calculations are further complicated in a system where chains of several transitions can occur within time t. The emergency brake system is a high level description and does not have chains of transitions to consider, making it possible to obtain the required probabilities as follows[5]:

$$p_{ij} = \begin{cases} \dfrac{\lambda_{ij}}{\sum\limits_{\forall k} \lambda_{ik}} \left(1 - e^{-\sum\limits_{\forall k} \lambda_{ik} t}\right) & i \neq j \\ \dfrac{\lambda_{ii}}{\sum\limits_{\forall k} \lambda_{ik}} \left(1 - e^{-\sum\limits_{\forall k} \lambda_{ik} t}\right) + e^{-\sum\limits_{\forall k} \lambda_{ik} t} & otherwise \end{cases}, \tag{4.2}$$

where p_{ij} is the probability that a transition from state i to state j occurs; and λ_{ij} is the rate at which transitions from state i to state j occur in the CTMC. As with the first option, p_{ij} is calculated for all possible state transitions to produce a DTMC, which can then be modelled with probabilistic choice statements in pB.

This approach allows for more powerful reasoning about the system, and would make it possible to determine dependability measures such as the mean time between failures. However, the required calculations are far from straightforward in the general case. The interpretation of any analysis results in terms of the parameters of the original CTMC is also not straightforward.

4.3.3 pB models and analysis

Separate models were created in pB for the emergency brake scenario for each of the abstraction options given in Section 4.3.2. In this section an overview of the models is provided and the analysis of the safety property (given in Section 4.1) is shown. Full descriptions of the pB models can be found in Appendix B.2, the proofs for the safety property can be found in full in Appendix B.3.

Both of the pB models[6] consist of two operations, *main* and *EB_Request*, to capture the behaviour of the emergency brake (see Figures 4.7 and 4.8). The *EB_Request* operation is the same in both models and describes what occurs after the emergency brake has been requested (either there is an unsafe failure with probability *p_us* or the brake is applied). The *main* operation models the choice between a safe failure occurring versus an emergency brake request. In option 2 there is also the possibility of just time

[5]Note that it is outside of the scope of this thesis to find a way of calculating such probabilities for the general case, as this is a complex problem in itself.

[6]Note that these models have been designed to be as comparable as possible to the Stochastic Event-B models later in the chapter, and thus pB's syntax has been slightly adapted to use the mathematical notation for probabilistic choice.

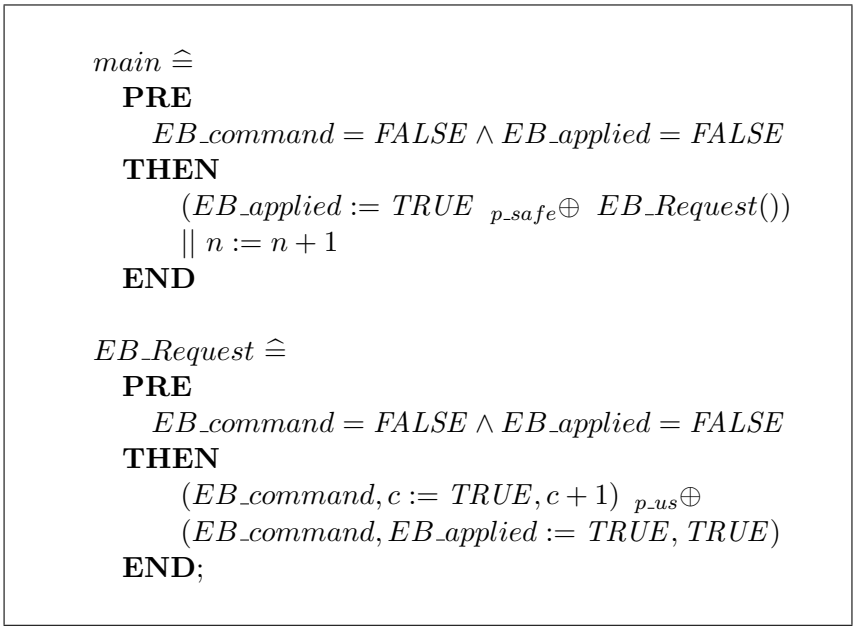

Figure 4.7: pB *main* and *EB_Request* operations for option 1

passing (without any other state change), which is modelled using $SKIP$. This enables the analysis of the rate at which events occur in the second model.

A safety analysis was carried out on both of the pB models. The safety property was modelled using the expectations clause described in Section 4.3.1. However, the formulation and analysis of such expectations requires access to historical data about the probabilistic behaviour to provide useful analysis. Thus *fresh variables* are required in the model to record: the number of times the model has been exercised; and of these times the number of occurrences of the property of interest. This is similar to the rewards approach of CTMCs, however in pB these fresh variables have to be integrated into the model.

For option 1 fresh variables n and c were added: n records the number of times the model has been exercised (i.e. is incremented every time *main* is called); and c records the number of unsafe failure occurrences (i.e. is incremented every time an unsafe failure occurs). These variables are used in an expectation, to record the safety property that the expected proportion of unsafe failures is bounded above (by some given value). This expectation is written as follows:

$$0 \leq \ n \times p\_max - c \ . \tag{4.3}$$

This is interpreted as $n \times p\_max - c$ is always at least 0, i.e. that the expected proportion of unsafe failures, $\frac{c}{n}$, is always at most some maximum p_max. Analysis of the above expectation determines how the probabilistic parameters (p_safe and p_us from Figure 4.7) relate to each other and the required maximum probability of unsafe failure

$main \mathrel{\widehat{=}}$
PRE
$EB\_command = FALSE \wedge EB\_applied = FALSE$
THEN
$(EB\_applied := TRUE \ {}_{p\_safe}\oplus \ (EB\_Request() \ {}_{p\_req}\oplus \ SKIP))$
$|| \ time := time + 1$
END

$EB\_Request \mathrel{\widehat{=}}$
PRE
$EB\_command = FALSE \wedge EB\_applied = FALSE$
THEN
$(EB\_command, c := TRUE, c + 1) \ {}_{p\_us}\oplus$
$(EB\_command, EB\_applied := TRUE, TRUE)$
END;

Figure 4.8: pB *main* and *EB_Request* operations for option 2

(p_max) in order to respect the expectation. The result (see Appendix B.3 for the full proof) obtained is:

$$(1 - p\_safe) \times p\_us \leq p\_max \ . \tag{4.4}$$

Informally, this states that the probability of an *EB_Request* happening and resulting in the unsafe failure state is at most p_max.

For option 2 fresh variables *time* and c were added: *time* records the number of time units that have passed (i.e. is incremented every time *main* is called); and c records the number of unsafe failure occurrences (i.e. is incremented every time an unsafe failure occurs). As before, these variables appear in the expectation to ensure that the rate of unsafe failures is bounded above by some given value. This expectation is written as follows:

$$0 \leq \ time \times p\_max - c \ . \tag{4.5}$$

This is interpreted as $time \times p\_max - c$ is always at least 0, i.e. that the expected rate of unsafe failures, $\frac{c}{time}$, is always at most some maximum p_max. Analysis of the above expectation shows how the probabilistic parameters (p_safe, p_req and p_us from Figure 4.8) relate to each other and the required maximum rate of unsafe failure (p_max) in order to respect the expectation. The result (see Appendix B.3 for the full proof) obtained is:

$$(1 - p\_safe) \times p\_req \times p\_us \leq p\_max \tag{4.6}$$

Informally, this states that the rate of an *EB_Request* (instead of a safe failure or $SKIP$, i.e. nothing) happening and resulting in the unsafe failure state is at most p_max. Note

that the result is given in terms of probabilities of events occurring within one time unit (forming an approximate rate of occurrence), as opposed to the rate parameters of the original CTMC. To get back to the rates it is required to substitute the p_{ij}'s according to the equations given in Section 4.3.2 and re-arrange/simplify the resulting inequality. This step is omitted here as it is not straightforward and does not contribute to the illustration of pB. The fact that this final step is arduous (especially for more complex systems) means that a discrete approximation is not a good approach for reasoning about continuous probability.

4.3.4 Experiences with pB

The probabilistic model-based specification language pB provides a good formal basis for the analysis of probabilistic systems. The proof-theoretic semantics of pB allow the user to rigorously analyse properties of interest. The proof rules can be used algebraically (as well as numerically) to derive properties of the model parameters with respect to the design requirements, as shown above. This enables the designer to gain a better understanding of the system, and analyse alternative designs at a higher level of abstraction. However, the designer must have a good understanding of the mathematics involved to do so, as this process is not automated (and is not simple to automate for the general case). A further advantage to pB (although not illustrated in this case study) is the use of abstraction and refinement. The refinement calculus [11, Preface] enables a designer to start with a simple abstract model and, through a series of (mathematically proven) refinements, develop a (more) concrete implementation of the abstract model. This process ensures that the correctness (i.e. the required properties) of the design is preserved in the final product.

The disadvantage of using a *probabilistic* model-based specification language (like pB) to model a system that is *stochastic* in nature has clearly been illustrated in this case study. There is an unavoidable trade-off between the complexity of the discrete approximation (and its analysis) and the information that can be obtained through its analysis. There are also some relatively trivial issues with pB: the tool support is not freely available, resulting in a cost overhead in using pB (either in terms of money or time to do manual proofs); the use of fresh variables to allow analysis is a little clumsy; and probabilistic choice clauses are restricted to two options, making multi-way probabilistic branching over complex.

Overall pB is an interesting approach to modelling and analysing stochastic behaviour. In particular the abstraction and refinement facilities, and the ability to obtain algebraic solutions, are very powerful and potentially very useful for designers. However, the lack of support for continuous probability distributions does limit its applicability somewhat as it is not really feasible for modelling systems that are naturally described using such distributions.

4.4 Modelling the Emergency Brake in Stochastic Event-B

This section prototypes a new stochastic model-based specification language (Stochastic Event-B) based on the Event-B formalism [3]. This aims to combine the benefits of model-based specification languages and formalisms that support continuous probability, as discussed in Sections 4.2 and 4.3. The analysis approach of Stochastic Event-B is similar to that of pB (using expectations to model properties of interest). However, it also includes support for modelling and analysing the exponential distribution (and potentially further continuous probability distributions). Event-B was chosen as the base language (instead of B) because Event-B has an open source toolkit[7] for proof support. Event-B also seems to be a more natural language for modelling the occurrence of failures, as behaviour is specified in *events* (instead of operations).

An overview of standard Event-B is given below, followed by a description of the proposed stochastic extensions. The emergency brake scenario is modelled and analysed in this proposed language as an illustration of its use. Stochastic Event-B is then evaluated and compared to the approaches discussed above.

4.4.1 Event-B overview

The Event-B formalism is derived from classical B [2], but also incorporates concepts from Action Systems [10]. The functionality of an Event-B model is given in *events*. Such events may update variables, but do not include flow constructs such as conditional choice, loops or even calls to other events. Instead the flow of an Event-B model is determined by a demonic choice over the set of events that may occur (see "guard" below) from the current state.

An Event-B model consists of two types of components: *machines* and *contexts*. A *machine* models the dynamic behaviour of the system such as the *variables* and the *events*. The *context* provides details of the static information – *constants*, *values* and *properties* over such values.

The *machine* description contains the bulk of the model and may include *variables*, *invariants* and *events*. Variables store the state of the machine. Invariants are used to constrain the types of the variables as well as state other logical properties over the variables that must hold at all times. Events define the behaviour of the system, i.e. the state transitions that may occur. Each event may include a *guard*, which defines the states from which the transition can occur, and includes a set *actions* which define the updates to the variables. For more detailed information about the contents of Event-B models the reader is referred to Abrial [3].

A number of *proof obligations* are automatically generated for an Event-B model. These state the requirements for the model to be internally consistent. For example, events must not invalidate the invariants (known as *invariant preservation*), the proof

[7] The Rodin platform, available at http://www.event-b.org/platform.html.

obligation that states this is as follows:

$$\begin{array}{l} \quad I(v) \\ \quad G(t,v) \\ \quad S(t,v,v') \quad , \\ \vdash \\ \quad I(v') \end{array}$$

where v and v' represent the variables of the machine before and after the event has occurred, respectively; t represents the parameters of the event; $I(v)$ represents the invariants over the variables; $G(t,v)$ represents the guard of the event; and $S(t,v,v')$ represents the actions occurring in the event (as a before-after predicate). Proof obligations are also used to define the refinement rules of Event-B, for details of these, and other proof obligations, the reader is referred to Abrial [3].

4.4.2 Stochastic extensions

As with classical B, standard reasoning in Event-B only determines what is possible, not what is probable. Such reasoning is not sufficient for analysing quantitative properties such as reliability and safety. Therefore some extensions to standard Event-B are proposed to support stochastic reasoning for the analysis of such properties.

Hallerstede and Hoang's work on introducing probabilistic to choice to Event-B [33] focuses on *qualitative* probabilistic reasoning; it is not possible to analyse numerical properties such as the emergency brake's safety requirement using such techniques. To reason about the safety requirement, *quantitative* reasoning is essential. Therefore it is proposed to extend Event-B actions with the probabilistic choice operator, i.e. to allow actions such as $x := 1 \ {}_p\!\oplus \ x := 2$.

The ability to model continuous probability distributions is also of importance. Two language extensions are proposed for this purpose. First it is proposed to allow events to occur according to the exponential distribution. To do this each event may have an associated *rate* parameter (written as a special kind of action called rate), which represents the rate of the occurrence of the event with respect to the exponential probability distribution. This is interpreted in a similar way to a PRISM (CTMC) command and enables CTMCs to be modelled in Stochastic Event-B. Note that events with rate parameters are selected probabilistically (like in CTMCs) as opposed to demonically.

The second construct allows more general probability distributions to be specified. It is proposed to extend Event-B actions to allow a variable to be assigned according to a continuous probability distribution. When used in conjunction with the rate parameters, this could be used to accumulate the amount of time that has passed in a CTMC for analysis purposes[8]. For example $time := time + \mathbf{exp}(\lambda)$ would increment a *time*

[8]Note that without the rate parameter events are chosen demonically, so for this situation both constructs are needed.

variable by a randomly assigned observation from the exponential distribution with parameter λ. However, it can also be used to model more general probabilistic behaviour, such as the level in a water tank (which may follow a normal distribution).

For analysis of stochastic properties, such as the safety property in the emergency brake scenario, the use of *expectations* as in pB (see Section 4.3.1) is advocated. Therefore, an expectations clause will be added to Event-B Machines to model these. The semantics of such statements is intended to be very similar to that of expectations in pB, except that Stochastic Event-B has *events* that would have to respect the expectations instead of pB's *operations*.

4.4.3 Event-B models and analysis

This section summarises how Event-B was used to model and analyse the case study. Three different models of the emergency brake are discussed. In a first model the limitations of analysing the scenario in standard Event-B (i.e. without any stochastic extensions) are demonstrated. Afterwards, two different options for analysing the stochastic behaviour of the emergency brake are explored. In option 1 time is modelled implicitly using a rate parameter. Option 2 makes use of the statement $time := time + \mathbf{exp}(\lambda)$ to update time explicitly. Note that these two options for modelling stochastic behaviour are semantically equivalent. The main differences are in the event and expectation notations, and the amount of flexibility each option provides, these issues are discussed towards the end of this section. Full descriptions of all the Event-B models can be found in Appendix B.4.

In standard Event-B the closest approximation to the emergency brake scenario involves a non-deterministic choice between possible events: EB_Normal, $Safe\_Failure$ and $Unsafe\_Failure$. There is no way of stating how often each of these events occur. Similarly, the best approach available for including the safety property is to model it as an invariant $EB\_command = TRUE \implies EB\_applied = TRUE$. The $Unsafe\_Failure$ event violates this invariant as it sets $EB\_command$ to $TRUE$, but $EB\_applied$ remains $FALSE$. Therefore, it can be concluded from the standard Event-B model that the safety property is not preserved by the $Unsafe\_Failure$ event. However, it is not possible to build an implementation of the emergency brake system in which it can be guaranteed 100% that an unsafe failure will *never* occur. Thus stochastic modelling is needed to establish and minimise the chances of an unsafe situation (and guarantee the stochastic version of the safety property).

For the Stochastic Event-B models of the emergency brake, the standard model described above is used as a basis and the EB_Normal and $Unsafe\_Failure$ events are combined into a single $EB\_Request$ event. This event includes a probabilistic choice statement that results in either the unsafe failure situation (with probability p) or the normal application of the emergency brake. Both the $EB\_Request$ event and the $Safe\_Failure$ event are assigned a *rate* value (λ_{req} and λ_{safe} respectively). These

Event $EB\_Request \mathrel{\widehat{=}}$

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

rate : λ_{req}

act1 : $(EB\_command, c, n := TRUE, c+1, n+1)$ $_p\oplus$
$(EB\_applied, EB\_command, n := TRUE, TRUE, n+1)$

end

Figure 4.9: Stochastic Event-B $EB\_Request$ event for option 1

Event $EB\_Request \mathrel{\widehat{=}}$

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

rate : λ_{req}

act1 : $(EB\_command, c := TRUE, c+1)$ $_p\oplus$
$(EB\_applied, EB\_command := TRUE, TRUE)$

act2 : $time := time + \mathbf{exp}(\lambda_{req} + \lambda_{safe})$

end

Figure 4.10: Stochastic Event-B $EB\_Request$ event for option 2

rate values are taken to be parameters of the exponential distribution parameter and model the rate at which the events occur. Note that this representation of the emergency brake scenario is analogous to the natural way of modelling the system, described in Section 4.1.

For the first Stochastic Event-B model considered (option 1), time is treated implicitly through the use of the *rate* parameter (see Figure 4.9). Similarly to the pB models, *fresh variables* are required for analysis, to track the history of the probabilistic choice statement. The total number of times the probabilistic choice is exercised is represented by n, and the number of times it resulted in an unsafe failure by c. The safety property is translated into the following expectation to be analysed:

$$0 \leq \ n \times \lambda_{max} - c \times \lambda_{req} \ . \tag{4.7}$$

This is interpreted as $n \times \lambda_{max} - c \times \lambda_{req}$ is always at least 0, i.e. that $\frac{c}{n} \times \lambda_{req}$ (the frequency of unsafe failures) always occurs at some maximum rate λ_{max}. The above expectation is analysed on the event of interest ($EB\_Request$) to provide a relationship

between the parameters of the model and the safety property. The following inequality is obtained as a result:

$$p \times \lambda_{req} \leq \lambda_{max} \ . \tag{4.8}$$

Informally, this means that the request rate of the brake, multiplied by the brake's probability of failure on demand, must be less than the allowed rate of unsafe failures. Full details of the expectation analysis can be found in Appendix B.5.1. Note that the expectation analysis for the $Safe\_Failure$ event is omitted, but is satisfied trivially as the event does not update any of the variables included in the expectation.

In the second Stochastic Event-B model considered (option 2), time is updated explicitly in the actions of the events according to the exponential distribution (see Figure 4.10). Notice that the time that passes before an $EB\_Request$ event occurs depends on both λ_{req} and λ_{safe}. From the definition of CTMCs, the distribution of the time to an $EB\_Request$ event is essentially the distribution of the sojourn time[9] of the initial state. The sojourn time of a state depends on the rate of all of the transitions available from that state. Intuitively the system can be thought to be delayed in a state for its sojourn time, before a probabilistic choice is made (according to the rate parameters) between the available transitions from that state.

The analysis of option 2 proceeds using just one *fresh variable* c (the number of unsafe failures), because time is being recorded explicitly. The safety property is translated into the following expectation to be analysed:

$$0 \leq \ time \times \lambda_{max} - c \ . \tag{4.9}$$

This is read as $time \times \lambda_{max} - c$ is always at least 0, i.e. that $\frac{c}{time}$ (the frequency of unsafe failures) always occurs at some maximum rate λ_{max}. The expectation in Formula 4.9 is analysed on the event of interest ($EB\_Request$) and gives the following

$$p \times (\lambda_{req} + \lambda_{safe}) \leq \lambda_{max} \ . \tag{4.10}$$

Full details of the expectation analysis can be found in Appendix B.5.2. Again, note that the expectation analysis for the $Safe\_Failure$ event is omitted, but is satisfied trivially as this event only increments $time$ (not c) and therefore never decreases the value of the expectation.

The analysis of option 2 provides an interesting result. Note how Formula 4.10 differs from the inequality found for option 1 (Formula 4.8), even though the two options are semantically equivalent. The reason for this discrepancy is due to how expectations work. These assume that the events are chosen demonically when multiple events are enabled from the same state. However, adding rate parameters to the events changes the demonic choice to a probabilistic choice, which is not taken into account in the analysis

[9]Recall that the sojourn time is defined as the time spent in a state before a transition occurs.

of expectations. In option 2, this leads to a conservative estimate of the relationship required between λ_{max} and the parameters of the model to satisfy the safety property. This feature of expectation analysis does not have any impact on the analysis of option 1, because the rate parameters used in that analysis are independent of the existence of other events (so the interpretation of how events are selected is irrelevant). This is not the case for the explicit time update found in option 2, which depends on the rates of all of the events available from the initial state.

Whilst this issue is caused by an inherent conflict in how expectations and rate parameters are interpreted, there is a workaround (for this specific case) to find the correct result. If the following expectation were analysed for option 2 (see Appendix B.5.2)

$$0 \leq \ time \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req} + \lambda_{safe}} \times c \ , \tag{4.11}$$

then the result would be the same as that given for option 1 (Formula 4.10). This explicitly incorporates the probability $\left(\frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}}\right)$ of the $EB\_Request$ event occurring (instead of the $Safe\_Failure$ event) into the expectation, and hence into the analysis.

The workaround used above requires knowledge of CTMCs and how the next event to occur is chosen probabilistically (according to the Embedded Markov Chain – see Section 4.3.2). However, this approach becomes more complex when the states of interest are reached from multiple events or chains of events. Therefore, this workaround is not recommended as a general solution to resolve the conflict between expectations and events that are selected probabilistically.

The second (explicit) approach to modelling the stochastic behaviour in Event-B is more flexible than the first (implicit) approach in terms of the behaviour that can be defined. However, care is needed when using expectations to analyse explicit assignments in events that are selected probabilistically instead of demonically.

4.4.4 Experiences with Stochastic Event-B

In this section the use of Stochastic Event-B for modelling the emergency brake case study is discussed. This includes some interesting observations about the analysis of expectations. Comparisons are also made to the PRISM and pB models discussed earlier in the chapter.

Both options for Stochastic Event-B, illustrated by the emergency brake case study above, provide a clean and useful method for combining stochastic and logical reasoning. The use of rates and the exponential distribution allow an intuitive model of the case study, whereas using probabilistic choice alone (as with pB) is rather more complicated. As a consequence the Stochastic Event-B models are more concise than those in pB (and require fewer proof steps[10] to analyse the safety property). With a more intuitive

[10]The pB models require nine and ten proof steps (for options 1 and 2 respectively) to discharge the expectation proof obligation, whereas the Stochastic Event-B models only take four and five (see Appendices B.3 and B.5).

notation, more complex problems should be easier to analyse. The expectation approach allows an algebraic analysis of the model (but must be used with care when events are selected probabilistically). This reveals the precise relationship between parameters of the model and the stochastic requirements, thus making the impact of design decisions more transparent. Finally, refinement rules could be established for Stochastic Event-B models allowing a formal and correct development from requirements to implementation.

Some valuable lessons were learnt whilst analysing the emergency brake scenario in the proposed stochastic version of Event-B. The first of these concerns the interaction between rate parameters and the use of expectations for analysis. Rate parameters make the choice of events probabilistic, whereas the expectation analysis approach assumes a demonic choice between events. When explicit stochastic assignments occur in events with rate parameters this can lead to conservative analysis of expectations. Thus care must be taken when performing expectation analysis in this situation. This issue was discussed in detail for the emergency brake case study (Section 4.4.3).

It would also seem that the way an expectation is formulated has an impact on the algebraic solution obtained. For example, for option 1, an alternative (semantically equivalent) expectation was initially analysed, $0 \leq \lambda_{max} - \frac{c}{n} \times \lambda_{req}$, and gave the result $p \leq \frac{c}{n}$. Whilst the two solutions are not contradictory (different variables of the model are referred to in each), the solution presented in Section 4.4.3 is clearly more useful for finding a suitable design for the system. Therefore the way in which expectations are formulated seems to impact on the usefulness of the results obtained.

There are a couple of other minor issues that a user should be aware of in the manipulation or simulation of an Event-B model with explicit stochastic assignments. There are rounding issues to consider when obtaining an observation from a continuous probability distribution, i.e. when simulating a statement such as $time := time + \mathbf{exp}(\lambda)$. Also the statement

$$(x := 1 \ {}_p\oplus \ x := 2) \ || \ time := time + \mathbf{exp}(\lambda)$$

would not be equivalent to

$$(x := 1 \ || \ time := time + \mathbf{exp}(\lambda)) \ {}_p\oplus \ (x := 2 \ || \ time := time + \mathbf{exp}(\lambda))$$

in general. This parallel substitution rule is valid for probabilistic programs. However, if there is a stochastic assignment on the right-hand side of || there needs to be some constraint in place to prevent different values being allocated to each instance of the stochastic assignment in the resulting statement. Note that this is not an issue if the analysis only depends on the *expected* value of the observation.

Overall, Stochastic Event-B looks like a promising approach to modelling and analysing stochastic systems. However, in order to define a full formal semantics for Stochastic Event-B there are many challenges to be overcome. These are discussed in the next

section.

4.5 Next Steps and Challenges

A stochastic version of Event-B promises to be a useful approach to modelling dependable systems, particularly those with naturally occurring continuous probability distributions. However, for Stochastic Event-B to become an accepted model-based specification language it requires a formal semantics. This is not a trivial problem to solve.

The key challenge in formally defining Stochastic Event-B is that demonic non-determinism is inherent in Event-B. The next event to occur in an Event-B model is chosen according to a demonic choice between all the events that are active in the current state. This non-determinism may be eliminated (in practice) from an Event-B model by including special variables to create a deterministic flow, or by using rate parameters[11]. However, the non-determinism is still inherent in the semantics of Event-B. This causes considerable challenge for a formal definition of the semantics of Stochastic Event-B, because (at a semantical level) the interaction between non-determinism and continuous probability is especially hard to resolve (see Chapter 7).

There is also very little literature on how continuous probability distributions could be included in *any* model-based specification language. Due to this and the above, it was decided to develop the formal semantics of a simpler stochastic model-based specification language (sGCL), instead of Stochastic Event-B. The language chosen for the basis of this new stochastic model-based specification language was pGCL (see Section 2.3.2). This is because constructs in pGCL are very simple, there are no functions or events for example, and because demonic non-determinism is modelled explicitly through the demonic choice operator. This operator could be removed and a formal deterministic stochastic language developed (Chapter 6), before exploring how non-determinism can be included as well (Chapter 7).

Even the semantics of a simple stochastic model-based language are challenging. The following issues are explored in the semantics of sGCL:

- How to define refinement in order to be as applicable as possible, whilst not being too restrictive?
- How do continuous probability distributions and demonic choice interact?
- How do recursion and continuous probability interact?
- How to deal with real numbers and truly infinite state spaces?

It is intended that the lessons learnt from developing a formal semantics of sGCL can be applied when reconsidering the formal definition of Stochastic Event-B. That is beyond

[11] But recall (Section 4.4.4) that this approach may cause other issues for expectation analysis.

the scope of this thesis, however.

4.6 Concluding Remarks

This chapter has used a simple case study (on an emergency brake) to illustrate the strengths and limitations of two contrasting approaches to modelling and analysing stochastic computer-based systems. The material presented in this chapter was not used as the sole basis for designing a stochastic model-based language, but rather was intended to illustrate key points of the discussion of the related work in Section 2.3. The prototype language introduced in this chapter (Stochastic Event-B) was used to illustrate the benefits of a stochastic model-based specification language, with respect to the case study. However, this language was not found to be suitable for a formal semantics definition, mainly because of the inherent demonic non-determinism in it. As will be seen in Chapter 7, integrating non-determinism and continuous probability in the semantics of a model-based specification language is far from a trivial problem. Therefore, the semantics of a relatively simpler language, sGCL, is formally defined in Part 2 of this thesis.

Part II

Defining sGCL

Chapter 5

Foundations of sGCL

This chapter provides some key theory and results that form the foundations of the semantic basis of a new language for describing computer-based systems with stochastic elements. This new language, called "Stochastic Guarded Command Language" or sGCL for short, is described in detail in Chapters 6 and 7.

An overview of measure theory is presented first (Section 5.1), providing many of the key definitions and terminology for the formal definition of continuous probability. This section includes an introduction to Lebesgue integration, a technique for integrating over (functions of) measures. A stochastic powerdomain, which forms the basis of sequential composition of stochastic programs, is then defined (Section 5.2). A further section describes a metric space for measures (Section 5.3), providing a basis for compactness arguments and recursion of stochastic programs. Finally, the semantics of pGCL (the version of GCL that enables discrete probability distributions, as summarised in Section 2.3.2) is described in enough detail to form a basis for the semantics of sGCL described in Chapters 6 and 7.

5.1 Measure Theory

Measure theory provides the mathematical foundation for all probability theory. It is the most general theory upon which all the well established results (such as those summarised in Section 2.1) are based. Measure theory is used as the semantic basis in sGCL partly because it allows the use of existing research (such as the stochastic powerdomain presented in Section 5.2), and partly because it makes the language as generally applicable as possible. This section summarises the measure theory results needed to understand the semantics of sGCL.

5.1.1 Basic definitions

A "measure" is essentially a function from a sets of values to a non-negative real numbers (often thought of as the size of the set). Only suitable subsets of a state space can be

measured. These are given by a "sigma algebra" (or σ-algebra) over that state space. A σ-algebra is defined as follows [7, Section 1.2.1]:

Definition 5.1. For state space S, a *σ-algebra* $\mathcal{A}$ over S is a collection of subsets of S that satisfies the following properties

1. $\mathcal{A} \neq \emptyset$,
2. $A \in \mathcal{A} \implies S \setminus A \in \mathcal{A}$,
3. $A_1, A_2, A_3, \ldots \in \mathcal{A} \implies A_1 \cup A_2 \cup A_3 \cup \cdots \in \mathcal{A}$. □

Together these three properties require that a σ-algebra contains at least $\emptyset$ and S, and that it is closed under countable unions and countable intersections. The elements of a σ-algebra are *measurable sets*. A pair $(S, \mathcal{A})$, where S is a state space and $\mathcal{A}$ is a σ-algebra over S, is known as a *measurable space*.

It is now possible to define a measure formally as follows [7, Section 1.2.3]:

Definition 5.2. For state space S and σ-algebra $\mathcal{A}$ over S, a *measure* μ is a function on $\mathcal{A}$ to the set of extended reals[1] that satisfies the following properties

1. $\mu(A) \geq 0$ for all $A \in \mathcal{A}$,
2. $\mu\left(\bigcup_i A_i\right) = \sum_i \mu\left(A_i\right)$ for all countable collections of disjoint sets $A_1, A_2, \ldots$ in $\mathcal{A}$,
3. $\mu(\emptyset) = 0$.

Note that the second property is known as *countable additivity*. □

The countable additivity property leads to some useful derived properties, for example monotonicity can be shown. Monotonicity states that $\mu(A_1) \leq \mu(A_2)$ for measure μ and measurable sets A_1, A_2 with $A_1 \subseteq A_2$. The triple $(S, \mathcal{A}, \mu)$ is known as a *measure space* for state space S, σ-algebra $\mathcal{A}$ and measure μ.

Three specific measures are of particular interest, "probability measures", "sub-probability measures" and "Lebesgue measure". A *probability measure* has the extra condition $\mu(S) = 1$ for state space S, measure μ. In a *sub-probability measure* the measure μ of the whole state space S is at most one, $\mu(S) \leq 1$. *Lebesgue measure* is the standard way of assigning length, area, etc. to Euclidean space, for example the Lebesgue measure on closed interval $[a, b]$ is equal to $b - a$ [7, Section 1.4].

5.1.2 Integration over measures

It is often necessary to integrate functions over measurable spaces, for example in calculating the expected value of a random variable. This section provides an overview of

[1] The extended real numbers is the set of reals plus $\pm\infty$.

"Lebesgue integration" and its relationship with the more commonly understood (Riemann) integral. Intuitively the Lebesgue integral can be thought to be dividing the area under the graph into slices according to the range of the function, whereas the Riemann integral does so according to the domain of the function. In doing so the Lebesgue integral provides a method for integration over spaces that are more general than Euclidean space. Lebesgue integration is used to define the weakest pre-condition semantics of stochastic assignment in sGCL. It is also used throughout the definition of the relational semantics of sGCL.

Before constructing the Lebesgue integral, it is instructive to define the collection of functions that can be integrated. These are called the "measurable functions" and are defined as [23, Section 4.2]

Definition 5.3. For measurable spaces $(S, \mathcal{A})$, $(T, \mathcal{B})$, a function f from S into T is *measurable* if $f^{-1}(B) \in \mathcal{A}$ for all $B \in \mathcal{B}$. □

The definition of the Lebesgue integral is built up from basic "indicator functions" and then generalised to more complex functions. An *indicator function*, written $I_A(x)$, is a function that returns one if $x \in A$, and zero otherwise. A *simple function* is a finite linear combination of indicator functions. The Lebesgue integral for a simple function is defined as [7, Section 1.5.3]:

Definition 5.4. For measure space $(S, \mathcal{A}, \mu)$ and simple function f, the Lebesgue integral of f with respect to μ is defined

$$\int_S f \, \mathrm{d}\mu = \sum_i a_i \mu(A_i) \ ,$$

where $f = \sum_i a_i I_{A_i}$ for reals a_i and disjoint sets A_i in $\mathcal{A}$. □

Note that an alternative notation for Lebesgue integration, $\int_\mu f$, is used in Chapters 6-8, where the state space S is defined in advance. Also, for more complex integrals, such as those in Section 5.2, the more explicit notation $\int_S f(x)\mu(\mathrm{d}x)$ is used to avoid confusion.

For more complex (non-negative) functions a limiting approximation is given to the Lebesgue integral as follows [7, Section 1.5.3]:

Definition 5.5. For measure space $(S, \mathcal{A}, \mu)$ and non-negative function f, the Lebesgue integral of f with respect to μ is defined

$$\int_S f \, \mathrm{d}\mu = \sup \left\{ \int_S s \, \mathrm{d}\mu \cdot s \text{ simple}, 0 \leq s \leq f \right\} \ .$$

□

Signed functions can be integrated as long as the integral of their absolute value is finite. This integral is defined as [7, Section 1.5.3]:

Definition 5.6. For measure space $(S, \mathcal{A}, \mu)$ and function f such that $\int_S |f| \, \mathrm{d}\mu < \infty$, the Lebesgue integral of f with respect to μ is defined

$$\int_S f \, \mathrm{d}\mu = \int_S f^+ \, \mathrm{d}\mu - \int_S f^- \, \mathrm{d}\mu \ ,$$

where $f^+ = max(f, 0)$ and $f^- = max(-f, 0)$. For example, $f^+(x) = f(x)$ when $f(x) > 0$, and 0 otherwise. □

It is worth noting that if a function is Riemann integrable on interval $[a, b]$, it can be shown [7, Section 1.7.1] that the function is also Lebesgue integrable on $[a, b]$, and that the two integrals are equal. This allows Riemann integration to be used instead of Lebesgue integration, if preferred, when a function is Riemann integrable.

5.2 A Stochastic Powerdomain

In its simplest form a stochastic program takes a state to a distribution over states. However, this representation leads to problems when finding the sequential composition of two stochastic programs as the input for the second program is now a distribution of states, not a single state. To overcome this problem powerdomains are used, these provide techniques for converting distributions to single states and vice versa. The theory behind such powerdomains comes from monads, in particular for a stochastic powerdomain the Giry monad is used [31, 22]. An overview of the Giry monad is described in this section, further details on monads in general can be found in standard texts on category theory and computation [66, 1]. The Giry monad provides the theory behind sequential composition in sGCL (Section 7.2).

Before the Giry monad is presented a few definitions are required. A *Polish space* is defined [22] as a separable metric space for which a complete metric exists. For the purposes of this thesis, it is suffice to know that the real number line is an important example of a Polish space, as are (open or closed) intervals of reals. The *Borel sets* of a state space S (written $\mathcal{B}_S$) include by definition [23] all of the open subsets of S and form a σ-algebra over that space. For example the Borel sets of the real number line includes all the intervals $(a, b]$ where $a, b \in \mathbb{R}$ [7, 23].

A monad is a triple $(T, \boldsymbol{\eta}, \boldsymbol{\mu})$ consisting of an "endofunctor" over a given category C, and two "natural transformations" [66]. The *endofunctor*, $T : C \to C$, is a functor that maps elements of the category to other elements of the same category. The first *natural transformation* (also known as *unit*), is defined as $\boldsymbol{\eta} : \mathbf{1}_C \to T$, where $\mathbf{1}_C$ represents the identity functor on C. The second natural transformation is defined as, $\boldsymbol{\mu} : T^2 \to T$, where $T^2 = T \circ T$. The transformations $\boldsymbol{\eta}, \boldsymbol{\mu}$ must satisfy the commutative diagrams shown in Figure 5.1. Informally, a monad represents a powerdomain where the natural transformations provide means of going up into and back out of the powerdomain.

For the Giry monad the endofunctor $\mathbb{S}$ takes a measurable space $(S, \mathcal{B}_S)$, where S is

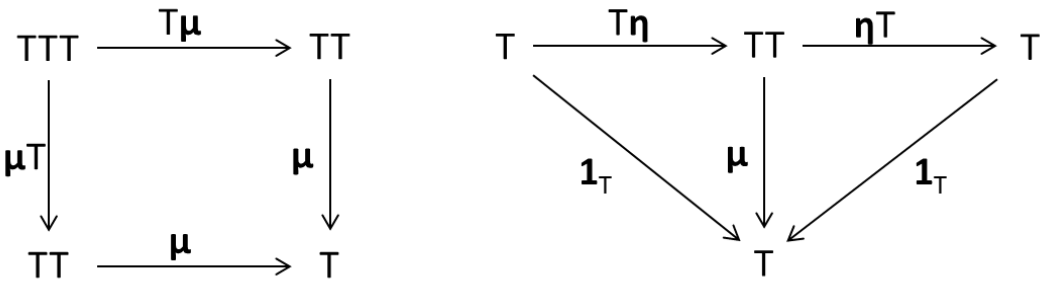

Figure 5.1: Monad coherence conditions represented as commutative diagrams [66]

a Polish space and $\mathcal{B}_S$ is the Borel sets of S, to the set of all sub-probability measures defined on that space (along with some suitable σ-algebra induced from $\mathcal{B}_S$). The unit transformation, $\boldsymbol{\eta}$, of the Giry monad allows states to be transformed into measures. It is based on the "Dirac measure", which is defined as [22]:

Definition 5.7. For a measurable space (X, Σ) and any measurable subset $A \in \Sigma$ the *Dirac measure* is defined

$$\delta_x(A) := \begin{cases} 1, & \text{if } x \in A, \\ 0, & \text{if } x \notin A, \end{cases}$$

for a given $x \in X$. □

Note that the Dirac measure can be used in a similar way to transform a single distribution into a distribution of distributions. The natural transformation $\boldsymbol{\eta}_S : S \to \mathbb{S}S$ of the Giry monad is simply the Dirac measure over measurable space $(S, \mathcal{B}_S)$. The second transformation $\boldsymbol{\mu} : \mathbb{S}^2 \to \mathbb{S}$ transforms measures into single states (or measures over measures into simple measures) by calculating the expected value and is defined as [22]:

Definition 5.8. The natural transformation $\boldsymbol{\mu}_S : \mathbb{S}\mathbb{S}S \to \mathbb{S}S$ is defined

$$\boldsymbol{\mu}_S(\mu)(A) := \int_{\mathbb{S}S} s(A)\mu(\mathrm{d}s)$$

for measurable space $(S, \mathcal{B}_S)$, initial measure μ over $\mathbb{S}S$ and final measurable set $A \in \mathcal{B}_S$. □

A Kleisli triple $(\mathbb{S}, \boldsymbol{\eta}, -^*)$ can be associated with the Giry monad $(\mathbb{S}, \boldsymbol{\eta}, \boldsymbol{\mu})$ defined above. Such a triple provides a morphism $-^*$ that converts a function $f : X \to \mathbb{S}Y$ to a function $f^* : \mathbb{S}X \to \mathbb{S}Y$. Informally, the $-^*$ provides a program with a mechanism for handling a measure as input when it would normally require a single state. It works by averaging the results for all elements of the state space with respect to the input measure. The composition of two stochastic programs can be defined according to the "Kleisli" product as [22]:

Definition 5.9. For stochastic programs $f : X \to \mathbb{S}Y$ and $g : Y \to \mathbb{S}Z$, the *Kleisli composition* $g^* \circ f$ is defined

$$(g^* \circ f)(x)(C) := \int_Y g(y)(C)\ f(x)(\mathrm{d}y)\ ,$$

for initial state $x \in X$ and final measurable set $C \in \mathcal{B}_Z$. □

Thus providing a means for the composition of two stochastic programs. Note that in the case where the stochastic programs are defined over the same state space S, the spaces X, Y and Z all reduce to S. This result will be used for defining the sequential composition of two stochastic programs in the relational semantics of sGCL (Section 7.2).

5.3 A Metric Space for Measures

When defining an iterative language with non-determinism in it, an appropriate way of determining compactness and convergence is required. To do this, there needs to be some way of determining how close two states are. When these states are actually distributions of states, this becomes more complex. Various metrics have been defined to compare distributions [30]. Of these, the "Kantorovich metric" is of particular interest when dealing with continuous probability and non-determinism [82]. This section provides the terminology required to understand the definition of the Kantorovich metric, before presenting the Kantorovich metric along with its duality and simplification for real numbers.

5.3.1 Joint probability distributions

The duality of the Kantorovich metric relies on the theory and terminology from joint probability distributions. A brief overview of the required knowledge is provided in this section. This builds on the terminology presented in Section 2.1.

A *joint probability distribution* describes the probability of pairs of (sets of) values instead of single (sets of) values. For both discrete and continuous functions the joint distribution can be characterised by its "joint distribution function" as follows [71]:

Definition 5.10. A *joint distribution function* over random variables X and Y is defined

$$F(x, y) := P(X \leq x, Y \leq y)\ .$$

□

The probability of observing a value within a specified rectangle can be found using the distribution function as follows

$$P(x_1 < X \leq x_2, y_1 < Y \leq y_2) = (F(x_2, y_2) - F(x_2, y_1)) - (F(x_1, y_2) - F(x_1, y_1))$$

for random variables X, Y and observations $x_1, x_2 \in X$ and $y_1, y_2 \in Y$.

The *joint mass function* and the *joint density function*, for joint discrete and continuous distributions respectively, relate in a similar way to the joint distribution function as described in Section 2.1. However, where single summations/integrations are required for (separate) distributions to convert from pmfs/pdfs to cdfs (see Section 2.1), double summations/integrations are required for joint distributions.

For a joint distribution over random variables X and Y it is often required to know the probability of a obtaining a certain value in X, without caring what the value of Y is. The distribution of X in this situation can be found by summing/integrating the pmf/pdf over all values of Y, for example

$$f_X(x) = \int_{-\infty}^{\infty} f(x, y) \, \mathrm{d}y \ ,$$

defines the *marginal density* of X for joint continuous distribution $f(x, y)$ over X, Y. Such a density function describes the *marginal distribution* of X (or *marginal measure* when defining distributions using measures). The marginal distribution of Y is defined symmetrically.

5.3.2 The Kantorovich metric

The Kantorovich metric provides an indication of how different two probability distributions are based on the "transportation problem" [83]. That is, if one imagines a probability distribution as a mound of earth, the Kantorovich metric calculates the optimal work involved in transforming one mound of earth to a differently shaped one. The Kantorovich metric is also known as the *Wasserstein metric* [30].

The Kantorovich metric is defined [20, Definition 2.1] as follows:

Definition 5.11. Given any two Borel probability measures μ and ν on separable metric space (S, d), the *Kantorovich distance* between μ and ν is defined by

$$K(\mu, \nu) := \sup \left\{ \left| \int_S f \mathrm{d}\mu - \int_S f \mathrm{d}\nu \right| \cdot ||f|| \leq 1 \right\} \ ,$$

where $|| \cdot ||$ is the *Lipschitz semi-norm* defined by $||f|| = \sup_{x \neq y} \frac{|f(x) - f(y)|}{d(x,y)}$ for a function $f : S \to \mathbb{R}$. □

The Kantorovich metric has a dual definition [83] for ease of understanding and calculation. This is defined [20, Definition 2.2] as follows:

Definition 5.12. Given any two Borel probability measures μ and ν on separable metric space (S, d), the metric L is defined by

$$L(\mu, \nu) := \inf \left\{ \int_{S \times S} d(x, y) \mathrm{d}\gamma(x, y) \cdot \gamma \in \Gamma(\mu, \nu) \right\} \ ,$$

where $\Gamma(\mu, \nu)$ is the set of all probability measures on $S \times S$ with marginal measures μ and ν. The probability measures μ and ν must satisfy $\int_S d(x, z)\mu(x) < \infty$ and $\int_S d(x, z)\nu(x) < \infty$ for all $z \in S$ respectively. Note that $\int_{S\times S}$ is used as a shorthand to mean the double integral $\int_S \int_S$. □

In both of these definitions $d(x, y)$ represents the distance between points x and y in the metric space over which the measures are defined. This is intuitively the cost associated with moving a single piece of dirt from x to y or vice versa.

A further simplification of the Kantorovich metric applies when the state space of the measures involved is the real number line, i.e. if the metric space (S, d) is the space of real numbers with Euclidean distance. This is defined [30] as follows:

Definition 5.13. Given any two Borel probability measures μ and ν on the metric space $(\mathbb{R}, d)$, where d represents Euclidean distance, the *Kantorovich distance* between μ and ν is defined by

$$K(\mu, \nu) := \int_{-\infty}^{\infty} |F(x) - G(x)| \ \mathrm{d}x \ ,$$

where $F(x)$ and $G(x)$ represent the cumulative distribution functions of μ and ν respectively. □

Note however that neither the duality, nor the simplification to real numbers, hold in general for *sub*-probability measures. So care must be taken over their use when working with sub-probability (see Appendix D.2).

5.4 Semantics of pGCL

The new stochastic language, sGCL, was developed using the probabilistic language pGCL [61] as a guide. Therefore, a closer look at the theory and semantics of pGCL is presented in order to aid the understanding of the semantics of sGCL. First, the description of the pGCL syntax and the transformer semantics (proof theory) given in Section 2.3.2 is reviewed, and where necessary more details are added. This is followed by a detailed description of the relational semantics (the underlying mathematics) of pGCL and the relationship between this and the transformer semantics. The healthiness conditions of pGCL are also discussed, these ensure that all programs written in pGCL have meanings.

In the following, the state space of pGCL, written S, is a finite set of states that programs may enter. Also, recall (Section 2.3.2) that the dot notation $f.x$ is used in pGCL to represent function application, $f(x)$.

	prog	*wp.prog.Q*
Abortion	abort	0
Identity	skip	Q
Assignment	$x := E$	$Q[x\backslash E]$
Composition	$prog_1; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$ else $prog_2$ fi	$[G]\times wp.prog_1.Q + [\neg G]\times wp.prog_2.Q$
Nondet. choice	$prog_1 \sqcap prog_2$	$wp.prog_1.Q \sqcap wp.prog_2.Q$
Probability	$prog_1\ {}_p\!\oplus\ prog_2$	$p{*}wp.prog_1.Q + (1{-}p){*}wp.prog_2.Q$
While-loop	do $G \to body$ od	$(\mathcal{F}X \cdot [G]\times wp.body.X + [\neg G]\times Q)$

x is a program variable; E is an expression in the program variables; $prog_1$ and $prog_2$ are probabilistic programs; G is a Boolean-valued expression in the program variables; p is a constant probability in $[0, 1]$; and Q is an expectation.

Given an expression Q, we write $Q[x\backslash E]$ to mean expression Q in which free occurrences of x have been replaced by expression E. $\mathcal{F}$ is the least fixed point operator w.r.t the ordering $\leq$ between expectations.

Scalar multiplication $*$, multiplication $\times$, addition $+$, subtraction $-$, minimum, $\sqcap$, and the comparison (such as $\leq$ and $<$) between expectations are defined by the usual point-wise extension of these operators as they apply to the real numbers. Multiplication and scalar multiplication have the highest precedence, followed by addition, subtraction, minimum and finally the comparison operators. Operators of equal precedence are evaluated from the left.

$[\cdot]$ is the function that takes a Boolean expression *false* to 0 and *true* to 1. For $\{0, 1\}$ real-valued functions, operation $\leq$ means the same as implication over predicates, and $\times$ represents conjunction. Addition over disjoint predicates is equivalent to disjunction.

Figure 5.2: Review of notation and weakest-precondition semantics of pGCL [6]

5.4.1 Syntax review

Recall (Section 2.3.2) that pGCL extends Dijkstra's Guarded Command Language [21] to include a probabilistic choice operator ${}_p\!\oplus$. For example the program fragment $x := 1\ {}_p\!\oplus\ x := 2$ means that x is assigned the value 1 with probability p, and 2 otherwise (with probability $1 - p$). The syntax of pGCL was described in Section 2.3.2 and is repeated in Figure 5.2 for convenience.

The syntax in pGCL has the usual meaning, but it is worth recalling that abort represents the program that can do anything, whilst skip represents the program that does nothing. An interesting feature of pGCL is that conditional choice is actually a special case of probabilistic choice. The program if *pred* then *prog* else *prog′* fi can be re-written as $prog\ {}_{[pred]}\!\oplus\ prog'$, where $[pred]$ returns one if predicate *pred* is true, and zero otherwise. Therefore any discussion and semantic definition of probabilistic choice also applies to conditional choice. It is also worth mentioning that the while-loop in pGCL is a special case of recursion, a more general repetition method. But as recursion in *sGCL* is limited to while-loops, there is no reason for a detailed discussion of the more general recursion available in pGCL. One final shorthand that is used in this thesis is *demonic assignment*, $x :\in \{expr_1, expr_2, \cdots\}$, which is interpreted as the demonic choice

$x := expr_1 \ \sqcap \ x := expr_2 \ \sqcap \ \cdots$.

5.4.2 Transformer semantics

The transformer semantics is the expectation[2] based approach for reasoning about pGCL programs as summarised in Section 2.3.2. A program is considered to transform (the expected value of) one random variable into another. This section provides a more formal description of these semantics.

First the space of transformers is defined [61, Definition 1.5.2] as follows:

Definition 5.14. The space of expectations over S is defined

$$\mathbb{E}S := (S \to \mathbb{R}_{\geq}, \leq) \ ,$$

where the relation $\leq$ is the pointwise extensions of the normal $\leq$ ordering in $\mathbb{R}_{\geq}$. The expectation-transformer model for programs is

$$\mathbb{T}S := (\mathbb{E}S \leftarrow \mathbb{E}S, \sqsubseteq) \ ,$$

where McIver at al. write the functional arrow backwards just to emphasise that such transformers map final post-expectations to initial pre-expectations, and where the refinement order $\sqsubseteq$ is derived pointwise from $\leq$ on $\mathbb{E}S$. □

This means that for one program to refine another the following [61, Definition 1.2.1] needs to hold:

Definition 5.15. A program $prog'$ is a refinement of a second program $prog$, written $prog \sqsubseteq prog'$ when for any post-expectation Q

$$wp.prog.Q \leq wp.prog'.Q \ ,$$

where $\leq$ represents the pointwise extension of $\leq$ between expectations.

It is worth noting at this point that if a program r satisfies $\alpha \leq wp.r.\beta$ then for every pair of distributions Δ_0, Δ in $\bar{S}$ such that Δ_0 can be taken to Δ by r the following should hold[3] [61, Formula 5.10]:

$$\sum_{\Delta_0} \alpha \ \leq \ \sum_{\Delta} \beta \ . \tag{5.1}$$

Informally, this means that the expectations must not decrease as execution proceeds.

The transformer semantics of the program constructs is given by the weakest precondition operator as shown in Figure 5.2.

[2] Also known as quantitative annotations or probabilistic predicates.

[3] Note that the sum notation is used throughout this definition of pGCL (instead of the integral notation adopted by McIver et al.) to distinguish it from the integration in the measure theory approach used in sGCL.

sublinearity	$c_1\,(t.\beta_1) + c_2\,(t.\beta_2) \ominus \underline{c} \leq \quad t.\,(c_1\beta_1 + c_2\beta_2 \ominus \underline{c})$
monotonicity	$\beta_1 \;\geq\; \beta_2$ implies $t.\beta_1 \;\geq\; t.\beta_2$
feasibility	$t.\beta \;\leq\; \sqcup\beta$
scaling	$t.\,(c\beta) \;\equiv\; c\,(t.\beta)$
continuity	$t.\,(\sqcup\beta) \;\equiv\; (\sqcup\beta : \mathcal{B} \cdot t.\beta)$

t is a transformer in $\mathbb{T}S$; β, β_1 and β_2 are expectations in $\mathbb{E}S$; c, c_1, c_2 are non-negative real numbers; $\mathcal{B}$ is a bounded, $\leq$-directed subset of $\mathbb{E}S$; and $\sqcup$ represents maximum. The symbol $\ominus$ represents subtraction in $\mathbb{R}_{\geq}$ so that $x \ominus y = (x - y) \sqcup 0$.

Figure 5.3: Healthiness conditions for pGCL transformers.

Healthiness conditions

The transformer model provides a rich language for specifying probabilistic programs. However, to enable some nice properties (such as "modular reasoning"[4]), and to ensure that all pGCL programs have meanings, the transformer model needs to be restricted through a set of "healthiness conditions".

The *healthiness conditions* required are "sublinearity", "monotonicity", "feasibility", "scaling" and "continuity". These are defined in Figure 5.3. McIver et al. show that if sublinearity holds, then monotonicity, feasibility and scaling also hold (i.e. these properties are derived from sublinearity) [61, Definition 1.6.2]. Continuity simplifies the treatment of recursion and enables the proof of the existence of a fixed point [61, Lemma 5.6.8]. Non-negativity of transformers is also an implicit healthiness condition, due to the restriction to non-negative expectations (see Definition 5.14).

5.4.3 Relational semantics

The relational semantics of pGCL gives a formal mathematical model as a basis to the language. This model of the language can then be compared to the transformer semantics to ensure that these two semantics are consistent and provide assurance that the proof rules are reasonable. The relational semantics of pGCL is described in detail in this section. First the semantics of deterministic probabilistic programs is defined and later built on to include non-determinism.

The main idea behind the relational semantics of pGCL is that a program takes an initial state S to a set of (discrete) probability distributions over the state space. A set of distributions is used instead of a single distribution to allow for non-determinism. The distributions are actually *sub-distributions* as the probabilities may sum to less than one, allowing for some chance of non-termination.

[4] McIver et al. use the term *modular reasoning* to mean that once a property of interest has been proven for a specific program, this property can be used in subsequent proofs about the program.

Deterministic relational semantics

For deterministic probabilistic programs a state maps to a single sub-distribution. McIver et al. [61, Definition 5.1.1] define a distribution[5] as:

Definition 5.16. For state space S, the set of *sub-distributions* over S is

$$\bar{S} = \{\Delta : S \to [0,1] \mid \sum \Delta \leq 1\} \ ,$$

the set of functions from S into the closed interval of reals [0,1] that sum to no more than one. (McIver et al. write $\sum \Delta$ to abbreviate $\sum_{s:S} \Delta.s$.) □

Note that Δ is a *total* function as unreachable final states are mapped to a probability of 0. A sub-distribution with a probability of 0 for every state represents the abort program.

McIver et al. [61, Definition 5.1.2] go on to define a complete partial order over distributions ($\bar{S}$):

Definition 5.17. For $\Delta, \Delta' \in \bar{S}$ define

$$\Delta \sqsubseteq \Delta' := (\forall s : S \cdot \Delta.s \leq \Delta'.s)$$

□

Thus the least element of $\bar{S}$ is when every state has a probability of 0 (i.e. the program with no chance of termination, abort), and $\bar{S}$ is maximal when the probabilities sum to one (i.e. a program with certain termination).

Using Definitions 5.16 and 5.17, the space of *deterministic* probabilistic programs over S and their refinement order can now be defined [61, Definition 5.1.3]:

Definition 5.18. For state space S the space of deterministic probabilistic programs over S is defined

$$\mathbb{D}S := (S \to \bar{S}, \sqsubseteq)$$

where for programs f, f' in $S \to \bar{S}$ McIver et al. define $\sqsubseteq$ pointwise:

$$f \sqsubseteq f' := (\forall s : S \cdot f.s \sqsubseteq f'.s) \ .$$

The order $\sqsubseteq$ of $\mathbb{D}S$ is called the *refinement* order. □

To complete the description of deterministic probabilistic programs it is important to show how (deterministic) standard programs can be defined in terms of them. McIver et al. [61, Definition 5.1.4] define a point distribution to assist in this goal:

[5] From here on distributions will be written to mean sub-distributions unless otherwise stated.

Definition 5.19. For state s the *point distribution* at s is defined

$$\bar{s}.s' := 1 \text{ if } (s = s') \text{ else } 0 \ .$$

□

Now the embedding of standard programs can be defined [61, Definition 5.1.5] as:

Definition 5.20. For every standard deterministic program f in $S \to S_\perp$ there is a corresponding deterministic program $\bar{f}$ in $\mathbb{D}S$, defined

$$\bar{f}.s := \begin{cases} \overline{f.s} & \text{if } f.s \neq \perp \\ \underline{0} & \text{otherwise,} \end{cases}$$

where s is the initial state. Equivalently McIver et al. define for arbitrary final state s' that

$$\bar{f}.s.s' := \begin{cases} 1 & \text{if } f.s = s' \\ 0 & \text{otherwise,} \end{cases}$$

noting in both cases that s, s' are restricted to proper elements (not $\perp$). □

A few more definitions are useful before the details of non-deterministic probabilistic programs can be defined. First a random variable is defined as [61, Definition 5.2.1]:

Definition 5.21. A *random variable* is a non-negative[6] real-valued function on the sample space which, for probabilistic programs, is the state space over which the programs operate. □

Using this definition the expected value of a probability distribution can be defined [61, Definition 5.2.2]:

Definition 5.22. For bounded random variable α in $S \to \mathbb{R}_{\geq}$ and distribution $\Delta \in \bar{S}$, the *expected value of* α *over* Δ is defined

$$\sum_{\Delta} \alpha := \sum_{s:S} (\alpha.s * \Delta.s) \ .$$

□

This definition of expected value provides a means for composing two probabilistic programs through a Kleisli composition as follows [61, Definition 5.2.3]:

Definition 5.23. For $f \in \mathbb{D}S$, (initial) distribution $\Delta \in \bar{S}$ and (final) state $s' \in S$ McIver et al. define f^*, an element of $\bar{S} \to \bar{S}$ as follows:

[6] The non-negative requirement on random variables is to ensure demonic choice can always be clear about the minimum value.

$$f^*.\Delta.s' := \sum_{\Delta}(f.s.s'\mathrm{d}s) \ ,$$

where $(f.s.s'\ \mathrm{d}s)$ is interpreted as a function over the state space S such that $s \in S$, i.e. it is equivalent to $\lambda s \cdot f.s.s'$. □

Non-deterministic relational semantics

The semantics for deterministic programs can now be extended to include demonic non-determinism, transforming the program space to approximately $S \to \mathbb{P}\bar{S}$. However, the set of distributions a demonic and probabilistic program can result in must be restricted as not all combinations are appropriate. Therefore the demonic probabilistic program space is restricted to the subset of distributions that satisfies "non-emptiness", "up closure", "convexity" and "Cauchy closure". *Non-emptiness* is trivially defined as requiring that the program results in at least one distribution, the rest are defined formally below.

Up closure is defined by McIver et al. [61, Definition 5.4.1] as follows:

Definition 5.24. A subset $\mathcal{D}$ of $\bar{S}$, a set of distributions, is *up closed* if it is closed under refinement of its elements – if for all $\Delta, \Delta' \in \bar{S}$ then

$$\Delta \in \mathcal{D} \text{ and } \Delta \sqsubseteq \Delta' \text{ implies } \Delta' \in \mathcal{D} \ .$$

□

This definition allows refinement to be expressed by reverse subset inclusion of the result sets, i.e. program r is refined by r' when $r'.s \subseteq r.s$ (for any initial state s). Note that the up closure of abort will be the whole of $\bar{S}$ as all programs refine abort.

Convexity is defined [61, Definition 5.4.2] as follows:

Definition 5.25. A set $\mathcal{D}$ of distributions is *convex* if for every $\Delta, \Delta' \in \mathcal{D}$ and probability $p \in [0, 1]$ then $\Delta \,{}_p\!\oplus\, \Delta' \in \mathcal{D}$ also. □

Requiring convexity allows a demonic choice of two programs to be refined by any probabilistic choice of the same two programs.

The definition of *Cauchy closure* requires a geometric interpretation of distributions. Each distribution $\Delta \in \bar{S}$ is considered a point in finite-dimensional Euclidean space, where each axis corresponds to an element s of S and the $s-$coordinate of distribution Δ is $\Delta.s$. Cauchy closure is then defined [61, Definition 5.4.3] as follows:

Definition 5.26. A set of distributions over S is *Cauchy closed* if as a subset of N-dimensional Euclidean space $\mathbb{R}^N$ it is closed in the usual sense,[7] where N is the (finite) cardinality of S. □

[7] That is, in the sense that it contains its boundary.

Finally the model for demonic probabilistic programs can be defined [61, Definition 5.4.4] as follows:

Definition 5.27. Given a state space S, the set of non-empty, up closed, convex and Cauchy closed subsets of $\bar{S}$ is written $\mathbb{C}S$, and such subsets are said to be *probabilistically closed*. It can be shown that $\mathbb{C}S$ is a *cpo* under $\supseteq$.

The *cpo* of demonic probabilistic programs over S is then defined

$$\mathbb{H}S := (S \to \mathbb{C}S, \sqsubseteq) \ ,$$

where for $r, r' \in \mathbb{H}S$

$$r \sqsubseteq r' := (\forall s : S \cdot r.s \supseteq r'.s) \ .$$

□

The relational semantics of the language constructs can now be defined. As an illustration the semantics of probabilistic choice, demonic choice and sequential composition is given below. For further details the reader is referred to He et al. [43].

Probabilistic choice is defined by McIver et al. [61, Definition 5.4.5] as follows:

Definition 5.28. For two programs $r, r' \in \mathbb{H}S$, their p-probabilistic combination is defined as

$$(r \ {}_p\!\oplus\ r').s := \{\Delta : r.s; \Delta' : r'.s \cdot \Delta \ {}_p\!\oplus\ \Delta'\} \ .$$

□

The relational semantics for demonic choice [61, Definition 5.4.6] is given by finding all possible probabilistic combinations[8] of the programs:

Definition 5.29. For two programs $r, r' \in \mathbb{H}S$, their demonic combination is defined as

$$(r \sqcap r').s := (\cup p : [0,1] \cdot (r \ {}_p\!\oplus\ r').s) \ .$$

□

The definition of sequential composition [61, Definition 5.4.7] relies on that of Kleisli composition (Definition 5.23):

Definition 5.30. For two programs $r, r' \in \mathbb{H}S$, their sequential composition is defined as

$$(r; r').s := \{\Delta : r.s; f : \mathbb{D}S \mid r' \sqsubseteq f \cdot f^*.\Delta\} \ .$$

[8] Note that this includes the cases where one of the programs is selected with probability one to allow $r \sqcap r' \sqsubseteq r$, for example.

where by $r' \sqsubseteq f$ McIver et al. mean $(\forall s : S \cdot f.s \in r'.s)$, agreeing with the notion of refinement that would result were $\mathbb{D}S$ to be embedded in $\mathbb{H}S$ in the obvious way. □

Therefore the sequential composition is found by taking all the possible intermediate distributions Δ that r can take from s, and finding for each the Kleisli composition of all the possible deterministic refinements f of r'.

5.4.4 Relating the transformer and relational semantics

To enable consistency checks between the two semantics of pGCL, relationships need to be defined between them. This section describes the relationship between the transformer and relational semantics of pGCL. In particular, the relational-to-transformer embedding and the transformer-to-relational retraction are given. Consistency of the transformer model with respect to the relational model is also discussed.

First, consider translating a model from the relational semantics to the transformer semantics. McIver and Morgan call this relational-to-transformer embedding [61, Definition 5.5.2] and define it formally as

Definition 5.31. The injection $wp \in \mathbb{H}S \to \mathbb{T}S$ is defined

$$wp.r.\beta.s := (\sqcap \Delta : r.s \cdot \sum_{\Delta} \beta) \ ,$$

for program $r \in \mathbb{H}S$, expectation $\beta \in \mathbb{E}S$, and state $s \in S$. □

Informally this states that the distribution that minimises the expected value of β is selected from the set of distributions in $\mathbb{H}S$. Minimum is used due to the demonic interpretation of non-determinism in pGCL.

The relationship from the transformer model to the relational model requires a little more work. This is because the transformer model is richer than the relational model and can thus describe programs that are invalid in the relational model. Therefore, a set of "regular transformers" is defined [61, Definition 5.5.3], which are exactly those that correspond to a valid relational model as follows:

Definition 5.32. The set of *regular expectations transformers* $\mathbb{T}_r S$ over S is the *wp*-image of $\mathbb{H}S$ in $\mathbb{T}S$, thus defined as

$$\mathbb{T}_r S := \{r : \mathbb{H}S \cdot wp.r\}$$

□

Note that $\mathbb{T}_r S$ is obtained when $\mathbb{T}S$ is restricted according to the healthiness conditions described in Section 5.4.2, i.e. $\mathbb{T}_r S$ is characterised by the healthiness conditions.

Now the inverse relationship, that of translating a model from transformer semantics into relational semantics, can be given. McIver and Morgan call this "transformer-to-relational retraction" [61, Definition 5.7.1], and define it as:

Definition 5.33. The function $rp \in \mathbb{T}S \to \mathbb{H}S$ is defined

$$rp.t.s := \{\Delta : \bar{S} \mid (\forall \beta : \mathbb{E}S \cdot t.\beta.s \leq \sum_{\Delta} \beta)\} \ ,$$

for transformer $t \in \mathbb{T}S$ and state $s \in S$. □

McIver et al. [61, Section 5.7] show that the two semantics are consistent by proving two results. The first is that (provided the domain of rp is restricted to $\mathbb{T}_r S$) there is a 1–1 correspondence between $\mathbb{H}S$ and $\mathbb{T}_r S$ through the mutual inverses wp and rp. The second is that the program constructs preserve the relationship between the transformer and the relational semantics [61, Section 5.8].

5.5 Concluding remarks

The theory presented in this chapter provides the building blocks for developing a stochastic model-based specification language. The new language, sGCL, is developed first as a deterministic language, and then the extensions required to make it non-deterministic are examined.

Measure theory is used throughout the definition of sGCL, whereas the material on the Giry monad and the Kantorovich metric is used in the definition of the relational semantics for the non-deterministic version of sGCL (a simpler existing theory can be used when there is no non-determinism present). The syntax and transformer semantics of pGCL are closely related to both versions of sGCL, whilst the relational semantics of pGCL was used as a guide throughout the development of the relational semantics of the non-deterministic version of sGCL.

Chapter 6

A Deterministic sGCL

McIver and Morgan's probabilistic language, pGCL [61], only allows discrete probability distributions to be included in a specification. This chapter presents a new *stochastic* GCL (sGCL), based on pGCL, that also allows assignment to continuous probability distributions (or *measures*). Demonic non-determinism is excluded from the language initially in order to simplify the presentation. It is re-introduced in Chapter 7.

The syntax (Section 6.2) and (proof-theoretic) transformer semantics (Section 6.3) of sGCL are presented. A measure theory representation of sGCL is briefly described (Section 6.4) and a unique link is provided between measure theory and the transformer semantics. The notions of refinement that are available in sGCL are also explored (Section 6.5). The chapter ends (Section 6.6) with a discussion of the language defined and how it compares to other stochastic specification languages. A working knowledge of the background material presented in Chapters 2 and 5 is assumed throughout this chapter.

6.1 Preliminaries

This section introduces the standard terminology and notation to be used throughout the definition and usage of sGCL. Some limitations of the state space are also set out.

The most important term to introduce is *measure*. The continuous probability in sGCL is defined using measure theory, therefore the term measure represents any continuous probability distribution (represented as a measure). To provide consistency with text books however, the term *distribution* is used when a named probability distribution, for example the uniform distribution, is being discussed. Although both terms represent a mapping from sets of states to probabilities, distributions tend to be defined in terms of probability density functions, and measures with respect to measure theory.

The state space of a program or measure will be denoted S. The state space is restricted to a metric space consisting of any bounded interval $[a, b]$ with standard Eu-

clidean distance[1], where a and b are finite real numbers. The restriction to bounded intervals allows the consistency of the transformer and relational semantics to be shown, but at some cost[2]. The σ-algebra (measurable subsets) of the state space of sGCL will be taken from the Borel sets of S. This will be denoted as $\mathcal{B}_S$ throughout. Any sub-interval of space $[a, b]$ is a Borel set of it, including the empty interval and the interval $[a, b]$ itself. For programs with multiple variables, the state space may be generalised to a bounded n-dimensional interval[3] (for finite n) such as $[a, b]^n$. Once more its Borel sets are used for the σ-algebra of this space.

The dot notation is used for function application throughout the definition and analysis of sGCL programs. This associates to the left so that $f.g.x$ is equivalent to $(f\,(g))\,(x)$[4]. This reduces the number of parentheses needed in the definitions and proofs, making it easier to parse. Finally the notation $:=$ is used to mean *is defined to be*.

6.2 Syntax

This section introduces the syntax of sGCL, which is essentially the same as that of pGCL (Section 5.4.1), but with an extra notation for assignment according to a measure. In the deterministic version of sGCL, defined in this chapter, demonic non-determinism is not carried across from pGCL. Some further syntactic sugar and restrictions on recursion are also discussed.

To assign a value to a (random) variable x according to a continuous probability measure μ the base syntax is defined:

$$x :\oplus \mu \ . \tag{6.1}$$

In the base syntax μ is a measure (Section 5.1) that maps the measurable subsets of the state space to some probability, formally, $\mu : \mathcal{B}_S \to [0, 1]$. The measure μ must satisfy all the properties of measures (see Section 5.1). Recall (Section 2.1) that continuous probability distributions have *infinite* state spaces as they are defined over the reals. This means that the probability of any single state is zero, probabilities have to be defined over subsets of the real line, hence the need for measure theory and the Borel sets of the state space. Note that μ is total because any unreachable sets of states would map to a probability of zero.

Measure theory (whilst being the most general approach) is not particularly intuitive

[1] The distance used for the metric space impacts on the range of values that the Kantorovich metric can take when finding the distance between two measures (see Appendix D for a more detailed discussion).

[2] This restriction excludes important distributions such as the exponential and Normal distributions. However, truncated (where the distribution is cut off at some finite limit and the remaining mass re-distributed proportionally) versions of these distributions can be used to approximate them if required.

[3] Also known as a hyperrectangle.

[4] Note that this is not the same as functional composition, $(f \circ g)(x)$, which is functional application that associates to the *right*.

or necessarily familiar to users of the formalism, so some syntactic sugar is also proposed for stochastic assignments. A first alternative is to define the measure according to its cumulative distribution function (see Section 2.1.2) and its bounds (using the keyword *cdf*):

$$x :\oplus \mathit{cdf}.F.\mathit{min}.\mathit{max} \tag{6.2}$$

where x is the (random) variable to be assigned to, F represents the cumulative distribution function and min and max represent the minimum and maximum values the distribution can take respectively. This notation has the advantage of being more commonly used in probability theory and it is also fairly easy to convert to the base (measure-theoretic) syntax as follows:

$$\mu.[a,b] := \begin{cases} F.b - F.a & \text{, for } \mathit{min} \leq a < b \leq \mathit{max} \\ 0 & \text{, otherwise} \end{cases} \tag{6.3}$$

where F, min and max are those given in the definition $\mathit{cdf}.F.\mathit{min}.\mathit{max}$.

In further syntactic sugar, a library of commonly used probability measures could be provided and declared as follows:

$$x :\oplus \mathit{distId}.\mathit{params} \tag{6.4}$$

where $distId$ is a unique identifier for the distribution and $params$ represents the parameters the distribution requires. For example the Normal distribution could be defined as $N.\mu.\sigma$ and the exponential distribution could be defined as $exp.\lambda$. These would then be converted to their measure theoretic definitions for analysis etc. It is not intended that a full library of measures be defined in this thesis, although some definitions may be given and used as shorthand for convenience in examples or the case study (Chapter 8).

At this point a discussion about recursion is required. The complexity of extending the state space to closed intervals of real numbers presents two interesting problems for recursion. The first issue is a problem with proving the continuity of recursion (thus allowing nested recursions). It is not possible to show in general that the limit of a sequence of continuous functions is continuous, for a state space of real numbers. In fact the following counterexample demonstrates this is not the case

$$f_n.x = \begin{cases} 0 & \text{if } x \leq 0 \\ n * x & \text{if } 0 < x < \frac{1}{n} \\ 1 & \text{if } x \geq \frac{1}{n} \end{cases} .$$

As n approaches infinity, the value of $\frac{1}{n}$ approaches zero, therefore $x \leq 0$ holds when x is zero as well as $x \geq \frac{1}{n}$. This means that the value of the limiting function at zero can be either zero or one, but nothing in between, thus creating a discontinuity. Therefore

recursion breaks a required healthiness condition of the language (see Section 6.3.1). The second issue relates to the fact that the state space has to be compact and as such is limited to closed intervals of the real space. An infinite loop has the potential to produce measures that break this. Consider, for example a fundamental theorem of statistics, the central limit theorem. This states that the sum of a sequence of independent and identically distributed distributions tends to a Normal distribution as the sequence tends to infinite length. The Normal distribution is not restricted to finite real numbers and as such is not allowed in sGCL. However, it would be easy to write a recursive program that returns the sum of an infinite sequence of independent and identically distributed distributions.

The resolution to these two problems is to restrict recursion in sGCL to finite (tail) recursion[5]. This makes the while-loop a base syntax construct for sGCL. The finiteness checks for while-loops are not built into sGCL, it is up to a user of the language to ensure that a loop terminates in a finite number of iterations in order for the sGCL analysis to be valid. In practice, finiteness would be proved using a loop variant that is initially finite and is guaranteed to decrease each iteration.

The syntax and transformer semantics of a deterministic sGCL is summarised in Figure 6.1. Recall (Section 5.4.1) that conditional choice is a special case of probabilistic choice, where p is a predicate $[pred]$.

6.3 Transformer Semantics

The transformer semantics of sGCL is based on that defined for pGCL (as described in Section 5.4.2). This section discusses how the transformer semantics of pGCL are changed to make them appropriate for the new language, a deterministic sGCL.

The first important change is to the expectation space $\mathbb{E}S$ (Definition 5.14), as this now needs to work over continuous probability measures. This now needs to transform a measurable space into another measurable space, thus including a σ-algebra for each of the state spaces as follows:

Definition 6.1. The space of expectations over state space S is defined

$$\mathbb{E}S := (S \to \mathbb{R}_{\geq}, \leq) \ ,$$

where the relation $\leq$ is the pointwise extension of the normal $\leq$ ordering in $\mathbb{R}_{\geq}$. The Borel sets $\mathcal{B}_S$ and $\mathcal{B}_{\mathbb{R}_{\geq}}$ of the spaces S and $\mathbb{R}_{\geq}$ respectively provide the σ-algebras to complete these measurable spaces. The expectation-transformer model for programs is

$$\mathbb{T}S := (\mathbb{E}S \leftarrow \mathbb{E}S, \sqsubseteq) \ ,$$

where the refinement order $\sqsubseteq$ is derived pointwise from $\leq$ on $\mathbb{E}S$. □

[5]Kozen [54] has a similar restriction to finite recursion in his probabilistic PDL.

	prog	*wp.prog.Q*
Abortion	abort	0
Identity	skip	Q
Assignment	$x := E$	$Q[x\backslash E]$
Stoch. assignment	$x :\oplus \mu$	$\int_\mu Q$
Composition	$prog_1; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$ else $prog_2$ fi	$[G]\times wp.prog_1.Q + [\neg G]\times wp.prog_2.Q$
Probability	$prog_1\ {}_p\!\oplus\ prog_2$	$p{*}wp.prog_1.Q + (1{-}p){*}wp.prog_2.Q$
While-loop	do $G \to body$ od	$(\mathcal{F}X \cdot [G]\times wp.body.X + [\neg G]\times Q)$

x is a program variable; E is a measurable function over the program variables; μ is a continuous probability measure; $prog_1$ and $prog_2$ are stochastic programs; G is a predicate over the program variables that results in a measurable set; p is a constant probability in $[0, 1]$; and Q is an expectation.

Given an expression Q, the meaning of $Q[x\backslash E]$ is the expression Q in which free occurrences of x have been replaced by expression E. $\mathcal{F}$ is the least fixed point operator w.r.t the ordering $\leq$ between expectations. Note that the while-loop is required to terminate in a finite number of iterations.

Scalar multiplication $*$, multiplication $\times$, addition $+$, subtraction $-$, and the comparison (such as $\leq$ and $<$) between expectations are defined by the usual point-wise extension of these operators as they apply to the real numbers. Multiplication and scalar multiplication have the highest precedence, followed by addition, subtraction, and finally the comparison operators. Operators of equal precedence are evaluated from the left.

$[\cdot]$ is the function that takes a Boolean expression *false* to 0 and *true* to 1. For $\{0, 1\}$ real-valued functions, operation $\leq$ means the same as implication over predicates, and $\times$ represents conjunction. Addition over disjoint predicates is equivalent to disjunction.

Figure 6.1: Syntax and weakest pre-condition semantics of deterministic sGCL

Note that because the state space is infinite, the expectations must be bounded above by some finite real number [61].

Similarly the comparison of expectations needs to change for continuous measures to use Lebesgue integration (Section 5.1). A program r satisfies $\alpha \leq wp.r.\beta$ if for every pair of measures μ, $\mu' \in \bar{S}$ where r takes μ to μ' the following holds:

$$\int_\mu \alpha \ \leq \ \int_{\mu'} \beta \ . \tag{6.5}$$

where $\int_\mu \beta$ is used as shorthand for the Lebesgue integral $\int_S \beta \ \mathrm{d}\mu$.

The semantics of the new stochastic assignment construct $(x :\oplus \mu)$ also needs to be defined. Therefore, the greatest pre-expectation semantics of stochastic assignment is defined as follows:

$$wp.\,(x :\oplus \mu)\,.Q := \int_\mu Q \tag{6.6}$$

where x is the variable being assigned to, μ represents a probability measure with state

space S, and Q represents the post-expectation of interest. Intuitively this gives the expected value of the random variable Q with respect to the probability measure μ over the state space S. Note that where the state space S is made up of several variables (not just x) the integral $\int_\mu Q$ is a partial integral with respect to x. Therefore the integration could result in an expression in terms of the other state variables instead of a constant value.

The transformer semantics of stochastic assignment and the other constructs is summarised in Figure 6.1 for convenience. Recall that demonic choice has been removed from this summary to create a deterministic sGCL.

The last definition required for the transformer semantics is that of the refinement ordering. This is defined as follows:

Definition 6.2. A program $prog'$ is a refinement of a second program $prog$, written $prog \sqsubseteq prog'$ when for any post-expectation Q

$$wp.prog.Q \leq wp.prog'.Q \ ,$$

where $\leq$ represents the pointwise extension of $\leq$ between expectations.

Intuitively, refinement can occur when there is some chance of non-termination (i.e. the probability of the state space S is less than one). A program $prog'$ refines another $prog$ when $prog'$ reduces the chance of non-termination, but doesn't reduce the probability of any other outcome occurring.

6.3.1 Healthiness of the transformer semantics

Before defining the relational semantics of a deterministic sGCL, it is important to sanity check (through a set of healthiness conditions) the transformer semantics presented here. The first property to show is *measurability*, that every program results in a measurable greatest pre-expectation when given a measurable post-expectation. Without this property sequential composition would not be possible as the greatest pre-expectation of the latter program becomes the post-expectation for the former (see Figure 6.1). The second property to show is *continuity* of the transformers, defined formally in the usual way in Definition 6.3 below

Definition 6.3. An expectation transformer $wp.prog$ is *boundedly continuous* iff the following holds

$$wp.prog.\,(\sqcup \mathcal{B}) \equiv (\sqcup Q : \mathcal{B} \cdot wp.prog.Q) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists. □

Continuity facilitates reasoning about loops using fixed-points. Transformers also need to be *non-negative*

$$wp.prog.Q \geq 0 \text{ whenever } Q \geq 0 \ ,$$

and *linear*

$$wp.prog.(a\alpha + b\beta) = a * wp.prog.\alpha + b * wp.prog.\beta \ ,$$

for expectations α, β and reals a, b. These two conditions together enable a unique link between the transformer semantics and measure theory (see Section 6.4).

The rest of this section is dedicated to demonstrating that the properties of measurability, continuity, non-negativity and linearity hold for all of the program constructs defined in Figure 6.1.

Measurability

Most of the program constructs trivially transform measurable functions into measurable functions using established rules about measurable functions [Chapter 3, Section 5][72]. However some of the program constructs require extra restrictions.

The transformer $wp.\mathsf{abort}.Q$ transforms any Q to the measurable function that maps every state to zero. Therefore abortion trivially respects measurability. Identity also trivially respects measurability as $wp.\mathsf{skip}.Q$ results in Q for any measurable function Q.

In order for assignment to respect measurability a further restriction is required. The transformer $wp.\,(x := E)\,.Q$ results in $Q[x\backslash E]$, i.e. the function Q where all free occurrences of x have been replaced by E. This only describes a measurable function for measurable Q if the expression E is also a measurable function. In practice, this means that E can be all the standard functions such as addition, multiplication of variables and constants as well as more complex functions such as trigonometric functions [72, 24].

Stochastic assignment trivially preserves measurability when Q is only a function of x in $wp.(x :\oplus \mu).Q$. In this situation the integral $\int_\mu Q$ results in a map from the state space to a (constant) real number representing the expected value of x under μ. The situation is more interesting when the state space is made up of several variables $x, y, z, \ldots$ and Q is a function of several of these. For example, consider the program

$$y :\oplus U.[0,2]; x :\oplus U.[0,1] \ ,$$

where $U.[a,b]$ represents the continuous uniform distribution on interval $[a,b]$. A property of interest may be the probability that $x \leq y$, represented by a post-expectation of $[x \leq y]$ where $[Q] = 1$ when Q holds, zero otherwise. Finding the probability $x \leq y$ requires the calculation

$$wp.\,(y :\oplus U.[0,2])\,.\,(wp.\,(x :\oplus U.[0,1])\,.[x \leq y]) \ .$$

Evaluating the transformer on the right gives $0 * [y < 0] + y * [0 \leq y \leq 1] + 1 * [y > 1]$, which is clearly a function of y[6]. But is it measurable? In this case it is because all of the sets involving y are measurable and y itself is measurable.

In the general case, stochastic assignment respects measurability for measurable Q due to the construction of the Lebesgue integral and the fact that the limit of a sequence of measurable functions is also measurable [72].

Sequential composition of two programs, $prog_1; prog_2$, trivially respects measurability as long as its constituent programs $prog_1$ and $prog_2$ do. If this is the case, the transformer $wp.prog_2.Q$ will result in some measurable function, say R, when Q is measurable. The transformer $wp.prog_1.R$ then has the measurable function R as its input, which is transformed into a further measurable function (through $prog_1$) as required.

The probabilistic choice of two programs $prog_1 \ {}_p\!\oplus \ prog_2$ satisfies measurability as long as $prog_1, prog_2$ do and p is a measurable function from S to $[0, 1]$. This is because the product and sum of measurable functions is also measurable. Conditional choice is a special case of probabilistic choice where $p = [G]$ for some predicate G. Conditional choice then also respects measurability as long as G represents a measurable set (and hence $\neg G$ does as the complement of a measurable set is also measurable). Any inequality, e.g. $f(x) \leq \alpha$ for some real number α and measurable function f over variable x, produces a measurable set.

When recursion is unfolded a sequence of functions is obtained, where the limit of the functions is the least-fixed point of the recursion. If the elements of the recursive function are measurable (e.g. the *body* and the guard $[G]$ in a while-loop), then each function in this sequence will also be measurable by the measurability of sequential composition and the fact that the sum of measurable functions is also measurable. Finally, the limit of a sequence of measurable functions is also measurable [72], therefore the least-fixed point of the recursion will be measurable and thus recursion (and while-loops as a special case) respect measurability for some measurable expectation Q.

To illustrate the discussion above, consider a counter example of a program respecting measurability. Assume that conditional choice does not require that the predicate G describes a measurable set. In this situation it is possible to write a program that results in a non-measurable greatest pre-expectation for some post-expectation as follows

$$\mathsf{if}\ G\ \mathsf{then}\ x :\oplus \bar{0}\ \mathsf{else}\ x :\oplus \bar{1}\ ,$$

where $\bar{a}$ represents the point distribution at a. Consider a post-expectation $Q = [x \leq 0]$ that returns one for input less than or equal to zero, and zero otherwise. The greatest pre-expectation for post-expectation Q is

[6] Note that if the left side is evaluated on post-expectation $0 * [y < 0] + y * [0 \leq y \leq 1] + 1 * [y > 1]$, the probability 0.75 is obtained as expected.

$$
\begin{aligned}
& [G] * wp.\,(x :\oplus \bar{0})\,.[x \leq 0] + [\neg G] * wp.\,(x :\oplus \bar{1})\,.[x \leq 0] \\
= \;& [G] * 1 + [\neg G] * 0 \\
= \;& [G]\ ,
\end{aligned}
$$

but $[G]$ is not measurable. Therefore G has to represent a measurable set for conditional choice to respect measurability.

Continuity

The continuity of each of the program constructs is demonstrated in full in Appendix C. These proofs are discussed here, most are straightforward but others require a more complex argument.

Abortion is trivially continuous (Lemma C.1) because abort has the same effect irrespective of the expectation. Identity is also trivially continuous (Lemma C.2) because skip leaves the expectation unchanged, and thus the maximum of a set of expectations.

Assignment is continuous because an assignment operation simply transforms the current state to any other valid state. As the ordering of the expectations is defined for any state, updating a state will not change the ordering of the expectations, and hence the supremum will not be changed under assignment. A similar argument follows for stochastic assignment, which is reinforced by the monotone convergence theorem [72] as shown in Lemma C.3.

The sequential composition of two programs $prog_1; prog_2$ is continuous as long as its constituent programs $prog_1$, $prog_2$ are (Lemma C.4), following a similar argument to that presented for measurability. The proof for the continuity of probabilistic choice is a bit more involved (Lemma C.5), relying on the fact that addition is continuous. However, continuity holds as long as both of the programs involved in the probabilistic choice are continuous (Lemma C.5). Conditional choice is also continuous, as a special case of probabilistic choice where the probability p is $[G]$ for some valid condition G.

Finally, consider the continuity of recursion. This is not possible to prove for recursion in general (see Section 6.2). However, it is possible to show that a finite while-loop is continuous as follows:

Lemma 6.4. A *finite* loop do $G \to$ *body* od is continuous when the program *body* is continuous, i.e.

$$
wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\,(\sqcup \mathcal{B}) \equiv (\sqcup Q : \mathcal{B} \cdot wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.Q)\ ,
$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists.

Proof. The proof follows from unfolding the definition of the loop, based on the fact that a finite loop terminates within n iterations for some finite n:

$$
\begin{array}{lll}
 & wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\,(\sqcup\mathcal{B}) & \\
\equiv & & \text{unfolding (up to } n \text{ iterations)} \\
 & \quad [\neg G] * (\sqcup\mathcal{B}) & \\
 & +\ \ [G] * wp.body.\,([\neg G] * (\sqcup\mathcal{B})) & \\
 & +\ \ [G] * wp.body.\,([G] * wp.body.\,([\neg G] * (\sqcup\mathcal{B}))) & \\
 & +\ \ \ldots & \\
 & +\ \ ([G] * wp.body)^n\,.\,([\neg G] * (\sqcup\mathcal{B})) & \\
\equiv & & G \text{ not dependent on } Q \\
 & \quad (\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
 & +\ \ [G] * wp.body.\,(\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
 & +\ \ [G] * wp.body.\,([G] * wp.body.\,(\sqcup Q : \mathcal{B} \cdot [\neg G] * Q)) & \\
 & +\ \ \ldots & \\
 & +\ \ ([G] * wp.body)^n\,.\,(\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
\equiv & & body \text{ and sequential composition continuous, } G \text{ not dependent on } Q \\
 & \quad (\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
 & +\ \ (\sqcup Q : \mathcal{B} \cdot [G] * wp.body.\,([\neg G] * Q)) & \\
 & +\ \ (\sqcup Q : \mathcal{B} \cdot [G] * wp.body.\,([G] * wp.body.\,([\neg G] * Q))) & \\
 & +\ \ \ldots & \\
 & +\ \ (\sqcup Q : \mathcal{B} \cdot ([G] * wp.body)^n\,.\,([\neg G] * Q)) & \\
\equiv & & \text{addition is continuous} \\
 & \left(\sqcup Q : \mathcal{B} \cdot \left(\begin{array}{ll} & [\neg G] * Q \\ +\ & [G] * wp.body.\,([\neg G] * Q) \\ +\ & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * Q)) \\ +\ & \ldots \\ +\ & ([G] * wp.body)^n\,.\,([\neg G] * Q) \end{array}\right)\right) & \\
\equiv & (\sqcup Q : \mathcal{B} \cdot wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.Q) & \text{folding (up to } n \text{ iterations)}
\end{array}
$$

where $([G] * wp.body)^n$ represents n nested applications of $[G] * wp.body.$ For example, $([G] * wp.body)^2\,.X$ is interpreted as $[G] * wp.body.\,([G] * wp.body.X)$.

□

As recursion in sGCL is restricted to finite while-loops this is sufficient.

Non-negativity

The program constructs of sGCL are examined to show that each transforms non-negative expectations to non-negative expectations. In most cases this is trivially true.

Abortion is trivially non-negative as $wp.\mathsf{abort}.Q$ takes all states to zero regardless of Q. Identity is also trivially non-negative whenever Q is non-negative as the expectation is unchanged by the skip program.

Both the standard and stochastic assignment transformers are non-negative for any Q that takes any valid state to a non-negative real (i.e. for any non-negative Q), because an assignment operation simply transforms the current state to any other valid state.

The sequential composition of two non-negative programs $prog_1, prog_2$ is non-negative. The transformer $wp.prog_2.Q$ results in some non-negative function R for any non-negative Q. This then forms the post-expectation for $prog_1$ and therefore

$$wp.(prog_1; prog_2).Q = wp.prog_1.(wp.prog_2.Q) = wp.prog_1.R$$

is non-negative because R is non-negative.

The probabilistic choice of non-negative programs $prog_1, prog_2$ is also non-negative. The transformer

$$wp.(prog_1 \ {}_p\!\oplus \ prog_2).Q = p * wp.prog_1.Q + (1-p) * wp.prog_2.Q$$

results in the sum of two non-negative functions, because p can only take values in $[0,1]$ therefore both p and $1-p$ must be non-negative. The sum of two non-negative functions, and therefore probabilistic choice, is non-negative. Conditional choice is a special case of probabilistic choice where only the values zero and one are possible for p, so non-negativity trivially holds for conditional choice as well.

Unfolding recursion produces a sequence of non-negative functions, where the limit of this sequence of functions is the least fixed point of the recursion. The limit of a sequence of non-negative functions is also a non-negative function, therefore recursion respects non-negativity. Consider the while-loop as a special case of recursion. The transformer semantics, $(\mathcal{F}X \cdot [G] * wp.body.X + [\neg G] * Q)$, of a while-loop for some non-negative Q is unfolded as follows:

$$\begin{array}{lll}
 & 0 & \\
\leq & [\neg G] * Q & Q > 0,\ [\neg G] \in \{0,1\} \\
\leq & & \text{unfolding} \\
 & [\neg G] * Q & \\
+ & [G] * wp.body.\,([\neg G] * Q) & \\
\leq & & \text{unfolding} \\
 & [\neg G] * Q & \\
+ & [G] * wp.body.\,([\neg G] * Q) & \\
+ & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * Q)) & \\
\leq & \ldots & \text{unfolding}
\end{array}$$

It can be seen that each unfolding step is greater than or equal to the first, which is non-negative when Q is non-negative. Therefore the limit of this sequence of functions must be greater than or equal to zero.

Linearity

The linearity of each of the program constructs is proved in full in Appendix C. The results are summarised here. Most of the program constructs are easily shown to be linear, but recursion requires more thought.

Abortion is trivially linear (Lemma C.6) because abort takes any expectation to zero and any linear combination of zeroes is also zero. It is also easy to show that identity is linear (Lemma C.7) due to the fact that expectations are unchanged by skip.

It is straightforward to show that assignment according to some expression is linear (Lemma C.8). This is because the substitution of a variable with a given expression distributes over addition. Stochastic assignment is trivially linear (Lemma C.9) because Lebesgue integration is linear.

For the sequential composition $prog_1; prog_2$, the linearity argument follows as per the previous healthiness conditions as long as $prog_1$ and $prog_2$ are linear (Lemma C.10). Likewise, the probabilistic choice of two programs is linear if its constituent programs are linear due to some simple algebra (Lemma C.11). Conditional choice is also linear, as a special case of probabilistic choice where the probability p is $[G]$ for some valid condition G.

Showing the linearity of recursion in general is hard, but it is possible to show that a finite while-loop is linear as follows:

Lemma 6.5. A *finite* loop $\mathsf{do}\ G \to body\ \mathsf{od}$ is linear when the program *body* is linear, i.e.

$$\begin{aligned} wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\,(a\alpha + b\beta) = \quad & a * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\alpha \\ + \quad & b * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\beta\ , \end{aligned}$$

for expectations α, β, reals a, b and linear programs $prog_1, prog_2$.

Proof. The proof follows from unfolding the definition of the loop, based on the fact that a finite loop terminates within n iterations for some finite n:

$$\begin{aligned} & wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\,(a\alpha + b\beta) \\ \equiv & \qquad\qquad \text{unfolding (up to } n \text{ iterations)} \\ & [\neg G] * (a\alpha + b\beta) \\ + \quad & [G] * wp.body.\,([\neg G] * (a\alpha + b\beta)) \\ + \quad & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * (a\alpha + b\beta))) \\ + \quad & \ldots \\ + \quad & ([G] * wp.body)^n\,.\,([\neg G] * (a\alpha + b\beta)) \end{aligned}$$

$\equiv$ simple algebra

$$
\begin{array}{ll}
 & a * [\neg G] * \alpha + b * [\neg G] * \beta \\
+ & [G] * wp.body.\,(a * [\neg G] * \alpha + b * [\neg G] * \beta) \\
+ & [G] * wp.body.\,([G] * wp.body.\,(a * [\neg G] * \alpha + b * [\neg G] * \beta)) \\
+ & \ldots \\
+ & ([G] * wp.body)^n\,.\,(a * [\neg G] * \alpha + b * [\neg G] * \beta)
\end{array}
$$

$\equiv$ *body* and sequential composition linear

$$
\begin{array}{ll}
 & a * [\neg G] * \alpha \\
+ & b * [\neg G] * \beta \\
+ & a * [G] * wp.body.\,([\neg G] * \alpha) \\
+ & b * [G] * wp.body.\,([\neg G] * \beta) \\
+ & a * [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \alpha)) \\
+ & b * [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \beta)) \\
+ & \ldots \\
+ & a * ([G] * wp.body)^n\,.\,([\neg G] * \alpha) \\
+ & b * ([G] * wp.body)^n\,.\,([\neg G] * \beta)
\end{array}
$$

$\equiv$ simple algebra

$$
\begin{array}{ll}
 & \left(\begin{array}{ll}
 & [\neg G] * \alpha \\
+ & [G] * wp.body.\,([\neg G] * \alpha) \\
+ & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \alpha)) \\
+ & \ldots \\
+ & ([G] * wp.body)^n\,.\,([\neg G] * \alpha)
\end{array}\right) * a \\
+ & \left(\begin{array}{ll}
 & [\neg G] * \beta \\
+ & [G] * wp.body.\,([\neg G] * \beta) \\
+ & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \beta)) \\
+ & \ldots \\
+ & ([G] * wp.body)^n\,.\,([\neg G] * \beta)
\end{array}\right) * b
\end{array}
$$

$\equiv$ folding (up to n iterations)

$$a * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\alpha + b * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\beta$$

where $([G] * wp.body)^n$ represents n nested applications of $[G] * wp.body.$ For example, $([G] * wp.body)^2\,.X$ is interpreted as $[G] * wp.body.\,([G] * wp.body.X)$.

□

As with continuity, this is sufficient because recursion is restricted to finite while-loops in sGCL.

6.4 Relational Semantics

A full relational semantics is not provided for a deterministic sGCL[7]. Instead, the "Riesz representation theorem" (Theorem 6.9) is used to demonstrate a unique relationship exists between the transformer semantics and measure theory. Therefore, the general form (and refinement ordering) of the relational semantics is described briefly, before providing the details of how the Riesz representation theorem links this to the transformer semantics. The refinement ordering is included because it is sometimes easier to reason about refinement in terms of measures instead of in terms of programs. The consistency of the relational refinement ordering with respect to the transformer refinement ordering is also proved.

The main idea behind the relational semantics is that a program takes an initial state to a probability measure over the state space. Sub-probability measures, where the total probability defined may be less than one, can also be used to allow for some chance of non-termination. Unlike pGCL, a probability measure (distribution) in sGCL is *continuous*, not *discrete*.

The definition of a continuous measure[8] requires the use of measure theory (Section 5.1). Recall (Section 6.1) that $\mathcal{B}_S$, the Borel sets of the state space S, will be used as the σ-algebra of the measure space. The Borel sets represent the measurable subsets of the state space. A measure is thus defined as follows:

Definition 6.6. For state space S, the set of *sub-probability measures* over S is

$$\bar{S} = \{\mu : \mathcal{B}_S \to [0,1] \mid \mu.\emptyset = 0 \wedge \textit{countably-additive}.\mu\},$$

the set of functions from $\mathcal{B}_S$ into the closed interval of reals [0,1] that are *countably additive* and assign zero probability to empty sets. □

Countable additivity is given the standard meaning in measure theory (Section 5.1) – for any set $A_1, ..., A_n$ of mutually disjoint subsets of $\mathcal{B}_S$, $\mu.(\bigcup_i A_i) = \Sigma_i \mu.A_i$ must hold. Note that Definition 6.6 does not explicitly restrict the probability of obtaining any value in state space S to less than one – this is implicit in the map $\mu : \mathcal{B}_S \to [0,1]$ as $\mathcal{B}_S$, being a σ-algebra, will contain the union of all the measurable sets of S within itself. This definition therefore gives the possibility for non-termination when the largest set in $\mathcal{B}_S$ maps to a value less than one. Finally note that μ is a *total* function from $\mathcal{B}_S$, any unreachable sets of states are mapped to zero probability, and as with pGCL a measure that maps every element of $\mathcal{B}_S$ to 0 represents the abort program.

For the purposes of refinement (discussed in detail in Section 6.5), the ordering over measures is now considered. This is similar to that of pGCL (Definition 5.17), but

[7]This is partly because the consistency proofs would be complex, but mainly because the Riesz representation theorem provides an alternative, and simpler, approach.

[8]The term "(probability) measures" will also encompass sub-probability measures unless otherwise specified

needs some adjustment as now the probabilities of *sets* of states need to be compared (as individual states have zero probability):

Definition 6.7. For $\mu, \mu' \in \bar{S}$ define

$$\mu \sqsubseteq \mu' := (\forall a \in \mathcal{B}_S \cdot \mu.a \leq \mu'.a) \ ,$$

where $\mathcal{B}_S$ represents the Borel sets of the state space S. □

This ordering essentially means that a measure μ' is only greater than another μ if the (graph of the) pdf (see Section 2.1.2) of μ' is everywhere greater than that of μ.

Some example pdfs are given in Figure 6.2 to illustrate how measure ordering works. The base measure (mu 1) in this example behaves like a uniform distribution over $[0, 1]$ for half of the time, and like abort the rest of the time. The measure (mu 2) that is being compared to the base measure varies in each graph to illustrate different situations. The first two graphs, (a) and (b), demonstrate that the similarity of the shape of the pdfs is not important as long as the two lines do not cross and all the pdfs integrate to no more than one. In both of these cases mu 2 is strictly greater than mu 1, as the line of mu 2 is above that of mu 1 for all values of x. Graph (c) gives an example where no ordering can be determined because the lines of the pdfs cross. Finally graph (d) shows that the two measures can range over different values and still have a defined ordering (as long as the domain of one measure is a superset of the other).

Measures form a complete partial order where the least element of $\bar{S}$ is when every measurable subset of the state space has zero probability (i.e. abort), and $\bar{S}$ is maximal when the pdf integrates to one (i.e. the maximal set of $\mathcal{B}_S$ maps to one).

From Definitions 6.6 and 6.7, the space of deterministic *stochastic* programs over S and their refinement order can now be defined:

Definition 6.8. For state space S the space of deterministic stochastic programs over S is defined

$$\mathbb{D}S := (S \to \bar{S}, \sqsubseteq)$$

where $\sqsubseteq$ is defined pointwise for programs f, f' in $S \to \bar{S}$ as:

$$f \sqsubseteq f' := (\forall s : S \cdot f.s \sqsubseteq f'.s) \ .$$

The order $\sqsubseteq$ of $\mathbb{D}S$ is called the *refinement* order. □

6.4.1 Linking the transformer and relational semantics

A unique link is now shown to exist between the measures described above and the transformers described in Section 6.3 using the Riesz representation theorem. This states that [72, Chapter 13, 4.23]:

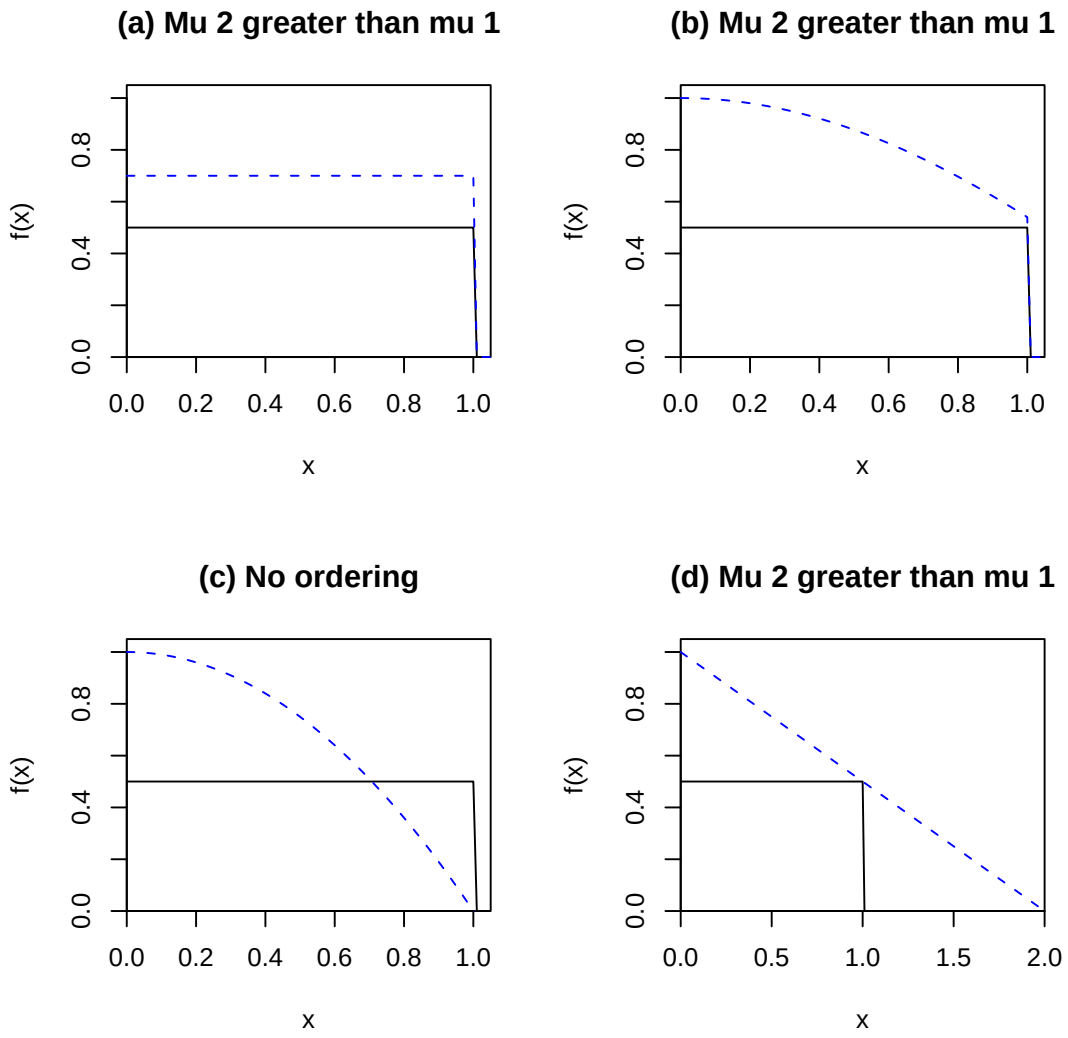

Figure 6.2: Examples of measure ordering: mu 1 (solid line); mu 2 (dashed line)

Theorem 6.9. Let X be a locally compact Hausdorff space and I a positive linear functional on $C_c(X)$. There is a unique Borel regular measure μ on X such that for all $f \in C_c(X)$

$$I(f) = \int_{\mu} f \ ,$$

where $C_c(X)$ is the space of continuous real-valued functions with compact support. □

Notice that the requirement of compact support causes issues if measures are allowed to range over the entire real line[9]. For example, consider a measure μ that is defined on $[0, \infty)$ and the post-expectation $[x > a]$ where x takes a value according to μ and a is a constant real number. This post-expectation returns one at infinity, not zero, therefore

[9] A function with compact support approaches zero as its input approaches infinity.

this function does not have compact support. This is why the restriction to measures over compact intervals has been enforced, otherwise a unique link between measures and the transformer semantics cannot be shown using the Riesz representation theorem (and may not even be possible to prove at all).

The other conditions of the theorem are also preserved. In Section 6.3.1 it was shown that linearity and non-negativity were preserved by the program constructs of sGCL. This satisfies the requirement of positive linear functionals. The state space of measures and expectations in sGCL is taken to be the real numbers, which is trivially a locally compact Hausdorff space. Finally, when X is a metric space (which trivially holds for real numbers), any finite Borel measure is regular [72]. All probability measures are finite, therefore this condition is satisfied by the measures used in sGCL.

Therefore, applying the Riesz representation theorem to the transformer semantics provides a unique representation of deterministic sGCL programs in measure theory.

The Riesz representation theorem can now be used to show that the transformer (Definition 6.2) and relational (Definition 6.7) definitions of refinement are equivalent. The equivalence is shown by proving each direction of the bi-implication separately. The first lemma and proof (Lemma 6.10) shows that a valid refinement in the transformer model is a valid refinement in the relational model.

Lemma 6.10. For sGCL programs $prog$, $prog'$ such that

$$prog \sqsubseteq prog' \ ,$$

there exist two unique measures μ, μ' that correspond to $prog$ and $prog'$ respectively and

$$\mu \sqsubseteq \mu' \ .$$

Proof. The proof follows using the Riesz representation theorem and by taking the post-expectation Q to be an indicator function over an arbitrary Borel set of the state space A:

$$
\begin{aligned}
& prog \sqsubseteq prog' \\
\implies \quad & wp.prog.Q \leq wp.prog'.Q && \text{Definition 6.2, arbitrary } Q \in \mathbb{T}S \\
\implies \quad & \int_{\mu} Q \leq \int_{\mu'} Q && \text{Riesz representation theorem (Theorem 6.9)} \\
\implies \quad & \int_{\mu} [x \in A] \leq \int_{\mu'} [x \in A] && \text{choose } Q \text{ to be } [x \in A] \text{ for arbitrary } A \in \mathcal{B}_S \\
\implies \quad & \mu.A \leq \mu'.A && \text{Lebesgue integration}
\end{aligned}
$$

□

The other direction (Lemma 6.11) shows that a valid refinement in the relational model is a valid refinement in the transformer model as follows:

Lemma 6.11. For measures μ, μ' such that

$$\mu \sqsubseteq \mu' \ ,$$

the corresponding programs *prog* and *prog*$'$ (to μ and μ' respectively) are such that

$$prog \sqsubseteq prog' \ .$$

Proof. The proof follows by defining the difference μ_δ of μ' and μ as $\mu_\delta.A := \mu'.A - \mu.A$ for all $A \in \mathcal{B}_S$ (for state space S). By its definition and Definition 6.7, the value of μ_δ must be non-negative for all $A \in \mathcal{B}_S$. The proof then proceeds for arbitrary $Q \in \mathbb{T}S$ and $s \in S$ as follows:

$$\begin{aligned}
& wp.prog.Q.s \\
\equiv \quad & \int_{\mu} Q && \text{Riesz representation theorem (Theorem 6.9)} \\
\leq \quad & \int_{\mu} Q + \int_{\mu_\delta} Q && \text{definition of } \mu_\delta \text{ and } \forall A \in \mathcal{B}_S \cdot \mu_\delta.A \geq 0 \\
\equiv \quad & \int_{\mu+\mu_\delta} Q && \text{linearity of integration} \\
\equiv \quad & \int_{\mu'} Q && \text{definition of } \mu_\delta \\
\equiv \quad & wp.prog'.Q.s && \text{Riesz representation theorem (Theorem 6.9)}
\end{aligned}$$

□

The equivalence of the two definitions of refinement then follows from Lemmas 6.10 and 6.11:

Lemma 6.12. For sGCL programs *prog*, *prog*$'$ such that

$$prog \sqsubseteq prog' \ ,$$

iff there exist two unique measures μ, μ' that correspond to *prog* and *prog*$'$ respectively and

$$\mu \sqsubseteq \mu' \ .$$

Proof. Follows directly from Lemmas 6.10 and 6.11 □

The equivalence of these two definitions allows the most suitable definition to be used for a given refinement proof. Some refinements are easier to show using the transformer definition, whilst others are easier using the relational definition. This is illustrated in the case study in Section 8.4.2.

6.5 Refinement Notions

In addition to the inherent refinement ordering in the definition of (deterministic) sGCL, a second notion of data refinement is also possible. This is particularly interesting for a language that includes both probabilistic choice and continuous probability distributions. A specification made up of a probabilistic choice statement could be refined by some expression over a continuous probability distribution that more closely represents the implementation. These two notions of refinement are explored in more detail below. Examples are used to illustrate each and some interesting issues are discussed.

6.5.1 Reducing non-termination

The ordering between probability distributions was discussed in some detail in Section 7.2.1. It is briefly revisited here for completeness. The idea behind the refinement ordering between continuous probability distributions is that these can be *sub-distributions*, where the total probability defined is less than one. The remaining probability represents the chance of the program not terminating. A refined program would have a lower chance of non-termination, thus having more of the probability allocated to valid states. However, the refined program is not allowed to have a lower chance of achieving an observation than its specification. This was defined formally in Definition 6.7. Graphically, this means that the pdf of the refining program cannot, at any state value, be lower than that of its specification. This was illustrated in Figure 6.2 (a more detailed explanation of these graphs can be found in Section 7.2.1).

To illustrate this notion of refinement, consider a simple program that assigns a value from a (truncated) Normal distribution[10] ninety percent of the time, but fails to terminate otherwise

$$x :\oplus N.0.1 \ {}_{0.9}\oplus \ \mathsf{abort} \ .$$

Such a program may represent, for example, the error of a sensor reading where the sensor concerned fails completely 10% percent of the time.

This could be refined by a program that assigns a value from the same truncated Normal distribution ninety nine percent of the time

$$x :\oplus N.0.1 \ {}_{0.99}\oplus \ \mathsf{abort} \ ,$$

representing a more reliable sensor that still produces an error according to a standard Normal distribution, but only fails completely 1% of the time.

However, it could not be refined by the program that assigns a value from a uniform

[10] The notation $N.\mu.\sigma$ is used to represent the Normal distribution with mean μ and standard deviation σ. It is assumed for compactness reasons that the distribution is truncated at some large, i.e. greater than one, but finite value. The details of the truncation are irrelevant for the purposes of this illustration, however.

distribution[11] over $[-1, 1]$ ninety nine percent of the time

$$x :\oplus U.[-1, 1] \ {}_{0.99}\oplus \ \mathsf{abort} \ ,$$

because this reduces the probability of obtaining values that are not in $[-1, 1]$.

6.5.2 Data refinement

Data refinement is also a very useful technique for designing and refining computer systems. It allows a designer to capture the essence of a program without having to know the exact data structures that will be used in the implementation. The details of the data structures to be used in the implementation can be included in formal refinements of the program. For example, a set of objects in which duplication and order is important may be represented abstractly by a list. In the actual implementation however, this data could be stored in an array, a linked list or something more complex. This thesis uses the term *data refinement* in a broader sense, as per McIver and Morgan [61]. They state that one datatype is refined by another if the second can replace the first in any program without detection, where only functional properties may be detected.

In a probabilistic or stochastic program there are some interesting opportunities for data refinement, as was illustrated by the steam boiler case study in pGCL [62]. In this study the steam boiler specification is the very abstract program

$$\mathsf{skip} \ {}_{p}\oplus \ \mathsf{abort} \ ,$$

that either works as intended (skip) with some probability, or fails (abort). The refinement steps then prove what the reliability of the regulator components must be in order to satisfy the specification.

Adding continuous probability distributions to the language provides further opportunities for data refinement. In sGCL a discrete probabilistic choice could be data refined by expressions over continuous probability distributions. For example, the abstract program

$$\mathsf{skip} \ {}_{0.9}\oplus \ \mathsf{abort}$$

represents a system that performs as expected with probability 0.9, and fails otherwise. This could be refined by a more concrete program such as

[11] The notation $U.[a, b]$ is used to represent the uniform distribution over the interval $[a, b]$.

$$
\begin{array}{ll}
x & :\oplus\ U.[0,1]; \\
 & \\
\textsf{if} & x > 0.9 \textsf{ then} \\
 & \quad \textsf{abort} \\
\textsf{fi} &
\end{array}
$$

The refinement above assigns a value according to a uniform distribution over $[0, 1]$, if the value is greater than 0.9 (which occurs with probability 0.1, as required for a valid refinement) the program aborts. This could represent the water level in a tank, for example, where anything over 0.9 is dangerously high. Data refinement in sGCL is discussed further in the case study (Chapter 8).

6.6 Discussion

This section discusses the strengths and limitations of sGCL as presented above and contrasts it with similar stochastic formalisms.

The main strength of sGCL is that it allows a variable to be assigned according to an arbitrary measure, not just the exponential distribution. However, as discussed in Section 6.4.1, the use of such a general construct raises semantic problems with the consequence that the state space has to be restricted to closed intervals. Therefore the sGCL language can not *even* assign a variable according to an exponential distribution. Although a truncated exponential distribution could be used as an approximation, this nullifies the memoryless properties of the exponential distribution. Note that the state space restriction is the reason why flash filestores are not revisited in the sGCL case study (Chapter 8), as neither of the distributions (exponential and Fréchet) occurring in flash filestores are defined over a closed interval. There are still many interesting problems that can be analysed, however, that are not covered by typical (Markov chain based) stochastic extensions of formal methods. One of these areas involves modelling the errors of sensor readings, which realistically could never be infinite anyway (unless the sensor does not return a value, which can be modelled by non-termination). In particular, with multiple sensor readings, sGCL provides the opportunity to analyse how these readings relate to each other in an elegant way. This allows, for example, the analysis of the frequency of some action that is triggered when two sensor readings drift too far apart. This example is explored in more detail in the case study (Chapter 8). A second restriction of sGCL is that infinite loops are not permitted. This disallows the typical (Markov chain based) analysis found in stochastic formal methods. However, as the exponential distribution can not be modelled in sGCL anyway, this further restriction has minimal impact. Nonetheless, it is an interesting opportunity for further work to determine if there exist other approaches to the semantics where these restrictions are not required.

The sGCL language was designed based on three main influences: Kozen's pioneering research on probabilistic languages [54]; Giry's research on stochastic monads [31]; and McIver and Morgan's probabilistic version of GCL [63, 61]. Therefore sGCL shares similarities with all of the above. The differences are briefly discussed below.

Kozen's probabilistic propositional dynamic logic (PDL) [54] shares many similarities with sGCL, particularly in the transformer semantics presented above. However, whilst his semantics covers continuous probability as well as discrete probability, the emphasis is on discrete probability. This means that the stochastic assignment operator (as given in sGCL) is not really discussed by Kozen. There is no expectation transformer definition for stochastic assignment and there are no examples or case studies showing its use in probabilistic PDL. The syntax of sGCL (particularly for probabilistic choice, which Kozen writes as a linear combination of programs) is considered to be more readable than that of probabilistic PDL. Kozen also does not prove the preservation of the healthiness conditions in his transformer semantics, although the properties of linearity and positivity are stated to hold in probabilistic PDL. But most importantly, Kozen does not consider sub-probability measures at all, as such no refinement ordering is given in probabilistic PDL. This is an interesting extension to Kozen's work that is discussed in more detail in Section 6.5 and Chapter 8.

Giry [31] describes a mathematical foundation for the sequential composition of measures based on monads. This stochastic powerdomain forms the basis of a relational semantics for stochastic programs. The deterministic version of sGCL does not make use of Giry's research, as it uses the Riesz representation theorem for the relational semantics instead. However, this is used in the non-deterministic version of sGCL described in Chapter 7 to define the Kleisli composition of sGCL programs. Giry's research does not consider an expectation transformer approach to complement the mathematical foundations.

McIver and Morgan [61] mainly focus on discrete probability through a probabilistic choice operator. Their language pGCL forms the basis of the sGCL language described above. However, in a paper on partial correctness Morgan and McIver [63] also discuss the extension to continuous probability. The approach proposed by them uses an alternative representation of distributions to measure theory. The alternative representation is essentially a discrete approximation known as "valuations" [25]. This alternative is not at all well known and as such was considered to be even more inaccessible to engineers than measure theory. Also, the body of research available for the relational semantics, such as the Riesz representation theorem (Theorem 6.9) and Giry's [31] stochastic powerdomain, rely on the use of measure theory to represent distributions.

Chapter 7

Towards a Non-Deterministic sGCL

Chapter 6 described a deterministic model-based specification language, sGCL, that allows a variable to be assigned according to a continuous probability measure. This chapter extends that definition of sGCL to include non-determinism. The extensions to the syntax, transformer semantics and healthiness conditions are straightforward (Section 7.1). Therefore, the main focus of the chapter is on defining a full relational semantics (Section 7.2) and on the challenge of proving the consistency between this and the transformer semantics (Section 7.3). The additional refinement notion available in a non-deterministic sGCL is explored in Section 7.4. The chapter ends with a discussion of the language extensions presented (Section 7.5).

This chapter assumes familiarity with the material on measure theory and domain theory presented in Chapter 5, and with the deterministic sGCL presented in Chapter 6.

7.1 Syntax and Transformer Semantics

The syntax and transformer semantics do not significantly change from the deterministic version of sGCL. The only difference is that (finite) demonic choice (as defined in pGCL [61]) is included. Note that the demonic choice is restricted to finite demonic choice, i.e. demonic choice over a finite state space. This is important because, even in standard programs, demonic choice over an infinite state space may break continuity [61, Section 8.4]. Recall (Section 6.3.1) that continuity is an important healthiness condition for sGCL.

The syntax and transformer semantics of a non-deterministic sGCL is summarised in Figure 7.1.

	prog	*wp.prog.Q*
Abortion	abort	0
Identity	skip	Q
Assignment	$x := E$	$Q[x\backslash E]$
Stoch. assignment	$x :\oplus \mu$	$\int_\mu Q$
Composition	$prog_1; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$ else $prog_2$ fi	$[G]\times wp.prog_1.Q + [\neg G]\times wp.prog_2.Q$
Nondet. choice	$prog_1 \sqcap prog_2$	$wp.prog_1.Q \sqcap wp.prog_2.Q$
Probability	$prog_1\ {}_p\!\oplus\ prog_2$	$p{*}wp.prog_1.Q + (1{-}p){*}wp.prog_2.Q$
While-loop	do $G \to body$ od	$(\mathcal{F}X \cdot [G]\times wp.body.X + [\neg G]\times Q)$

x is a program variable; E is a measurable function over the program variables; μ is a continuous probability measure; $prog_1$ and $prog_2$ are probabilistic programs; G is a predicate over the program variables that results in a measurable set; p is a constant probability in $[0, 1]$; and Q is an expectation.

Given an expression Q, the meaning of $Q[x\backslash E]$ is the expression Q in which free occurrences of x have been replaced by expression E. $\mathcal{F}$ is the least fixed point operator w.r.t the ordering $\leq$ between expectations. Note that the while-loop is required to terminate in a finite number of iterations.

Scalar multiplication $*$, multiplication $\times$, addition $+$, subtraction $-$, minimum $\sqcap$, and the comparison (such as $\leq$ and $<$) between expectations are defined by the usual point-wise extension of these operators as they apply to the real numbers. Multiplication and scalar multiplication have the highest precedence, followed by addition, subtraction, minimum and finally the comparison operators. Operators of equal precedence are evaluated from the left.

$[\cdot]$ is the function that takes a Boolean expression *false* to 0 and *true* to 1. For $\{0, 1\}$ real-valued functions, operation $\leq$ means the same as implication over predicates, and $\times$ represents conjunction. Addition over disjoint predicates is equivalent to disjunction.

Figure 7.1: Syntax and weakest pre-condition semantics of non-deterministic sGCL

7.1.1 Healthiness conditions

When demonic choice is included in a language, linearity no longer holds. Therefore the healthiness conditions defined in Section 6.3.1 need to be revisited. However, the weaker condition of sublinearity can be used instead. Therefore, non-deterministic sGCL programs have the healthiness conditions of measurability, continuity, non-negativity and sublinearity. *Sublinearity* is defined below in Definition 7.1, the other conditions are as defined in Section 6.3.1.

Definition 7.1. An expectation transformer *wp.prog* is *sublinear* iff for all $\alpha, \beta \in \mathbb{E}S$ and $a, b, c \in \mathbb{R}_{\geq}$ the following holds

$$wp.prog.\,(a\alpha + a\beta \ominus c) \geq a * wp.prog.\alpha + b * wp.prog.\beta \ominus c\ .$$

where $x \ominus y$ means $(x - y) \sqcup 0$. □

The condition of sublinearity is also in pGCL. Therefore it is only required to show that the new stochastic assignment operator respects this new condition. This is shown in

Lemma 7.2 below.

Lemma 7.2. The program $(x :\oplus \mu)$ is sublinear, i.e.

$$wp.\,(x :\oplus \mu)\,.\,(a\alpha + b\beta \ominus c) \geq a * wp.\,(x :\oplus \mu)\,.\alpha + b * wp.\,(x :\oplus \mu)\,.\beta \ominus c\ ,$$

for expectations α, β and non-negative reals a, b, c.

Proof.

$$\begin{aligned}
& wp.\,(x :\oplus \mu)\,.\,(a\alpha + b\beta \ominus c) \\
\equiv\ & \int_\mu (a\alpha + b\beta \ominus c) && wp \text{ definition} \\
\equiv\ & a\int_\mu \alpha + b\int_\mu \beta \ominus c\int_\mu 1 && \text{linearity and monotonicity of Lebesgue integration} \\
\geq\ & a\int_\mu \alpha + b\int_\mu \beta \ominus c && \mu \text{ a sub-probability measure implies } \textstyle\int_\mu 1 \leq 1 \\
\equiv\ & a * wp.\,(x :\oplus \mu)\,.\alpha + b * wp.\,(x :\oplus \mu)\,.\beta \ominus c && wp \text{ definition}
\end{aligned}$$

□

It is also important to check that the additional construct of demonic choice respects the healthiness conditions. Non-negativity and continuity are conditions of pGCL, so demonic choice has already been shown to respect these [61]. Measurability is a new condition for sGCL however, therefore this needs to be shown for demonic choice. It is trivially true from the properties of measurable functions. The demonic choice of two measurable functions is measurable because the (pointwise) minimum of two measurable functions is also measurable [12, Chapter 3, Section 26].

7.2 Relational Semantics

The full relational semantics of sGCL needs to be defined when non-determinism is introduced as the Riesz representation function no longer provides a 1-1 link between the transformer and relational semantics. This section starts by defining a relational semantics for deterministic sGCL, before building on those definitions to define the relational semantics of a non-deterministic sGCL.

7.2.1 Deterministic relational semantics

Recall (Section 6.4) that the main idea behind the relational semantics is that a program takes an initial state to a sub-probability measure over the state space (Definition 6.6):

Definition. For state space S, the set of *sub-probability measures* over S is

$$\bar{S} = \{\mu : \mathcal{B}_S \to [0,1] \mid \mu.\emptyset = 0 \wedge \textit{countably-additive}.\mu\},$$

the set of functions from $\mathcal{B}_S$ into the closed interval of reals [0,1] that are *countably additive* and assign zero probability to empty sets. □

Recall also, that the state space of measures, using the refinement ordering given in Definition 6.7, is:

Definition. For state space S the space of deterministic stochastic programs over S is defined

$$\mathbb{D}S := (S \to \bar{S}, \sqsubseteq)$$

where $\sqsubseteq$ is defined pointwise for programs f, f' in $S \to \bar{S}$ as:

$$f \sqsubseteq f' := (\forall s : S \cdot f.s \sqsubseteq f'.s) \ .$$

The order $\sqsubseteq$ of $\mathbb{D}S$ is called the *refinement* order. □

The rest of the relational semantics for a non-deterministic sGCL builds on these definitions.

For the purposes of representing standard (non-probabilistic) programs within sGCL, the *point distribution* needs to be defined. Such a distribution selects a single value with probability one (see Definitions 5.19 and 5.20). The Dirac measure (Section 5.1) is used to define a stochastic point distribution as follows:

Definition 7.3. For state $s \in S$ and measurable set $A \in \mathcal{B}_S$ the *stochastic point distribution* is defined

$$\bar{s}.A := 1 \text{ if } s \in A \text{ else } 0 \ .$$

□

For example, a standard state of 2 would be written as $\bar{2}$, where $\bar{2}.A$ returns one when 2 is an element of the set A and zero otherwise.

The embedding of standard programs in sGCL is given using the stochastic point distribution as follows:

Definition 7.4. For every standard deterministic program f in $S \to S_\perp$ there is a corresponding stochastic deterministic program $\bar{f}$ in $\mathbb{D}S$, defined

$$\bar{f}.s := \begin{array}{ll} \overline{f.s} & \text{if } f.s \neq \perp \\ \underline{0} & \text{otherwise,} \end{array}$$

where s is the initial state. Equivalently the embedding is defined for an arbitrary measurable subset $A \in \mathcal{B}_S$, as

$$\bar{f}.s.A := \begin{array}{ll} 1 & \text{if } f.s \in A \\ 0 & \text{otherwise,} \end{array}$$

noting in both cases that s is restricted to proper elements (not $\perp$). □

It is also of interest to be able to embed discrete probability distributions into the continuous probability model. Consider a discrete probability distribution Δ such that $\Delta.s_1 = p_1$, $\Delta.s_2 = p_2$, ..., $\Delta.s_n = p_n$. This can be represented in sGCL by using the point distribution definition above as follows:

Definition 7.5. For discrete distribution Δ such that $\Delta.s_1 = p_1$, $\Delta.s_2 = p_2$, ..., $\Delta.s_n = p_n$ and measurable set $A \in \mathcal{B}_S$ the continuous equivalent $\overline{\Delta}$ is defined:

$$\overline{\Delta}.A = \sum_{i=1}^{n} p_i * \overline{s_i}.A \ ,$$

where $s_i \in S$, $p_i \in [0,1]$ and $\sum_i p_i \leq 1$. □

This conversion can then be used to transform discrete probabilistic programs into *continuous* ones in the same way as for standard programs above.

Now consider the the definition of random variables, expectation and Kleisli composition for continuous probability measures.

The expected value of a *continuous* probability measure can be defined as:

Definition 7.6. For random variable α in $S \to \mathbb{R}_{\geq}$ and measure $\mu \in \bar{S}$, the *expected value of α over μ* is defined

$$\int_{\mu} \alpha := \int_{S} \alpha \ \mathrm{d}\mu \ .$$

□

The Kleisli product allows the composition of two stochastic programs. This involves *lifting* a stochastic program to allow it to be applied to a measure instead of a single state as shown in Definition 7.7. This is based on the Giry monad (see Section 5.2) for converting measures to states and vice versa.

Definition 7.7. For $f \in \mathbb{D}S$, (initial) measure $\mu \in \bar{S}$ and (final) set of states $A \in \mathcal{B}_S$ define f^*, an element of $\bar{S} \to \bar{S}$, as follows:

$$f^*.\mu.A := \int_{\mu} (f.s.A \ \mathrm{d}s) \ ,$$

where $(f.s.A \ \mathrm{d}s)$ is interpreted as a function over the state space S such that $s \in S$, i.e. it is equivalent to $\lambda s \cdot f.s.A$. □

The Kleisli composition of g after f can then be given by $g^* \circ f$ (as shown in Definition 5.9). This forms the basis for the definition of sequential composition given in Definition 7.15.

7.2.2 Non-deterministic relational semantics

The main idea behind the non-deterministic relational semantics is that a program takes an initial state to a set of probability measures over the state space. A set of measures is used instead of a single measure to provide non-determinism. The program space then becomes approximately $S \to \mathbb{P}\bar{S}$, where $\bar{S}$ is as defined in Definition 6.6. However, the set of measures a demonic and stochastic program can result in must be restricted as not all combinations are appropriate.

The demonic stochastic program space is restricted to the subset of measures that satisfies "non-emptiness", "up closure", "convexity" and "compactness". Note that these are almost the same restrictions as were found in pGCL, but Cauchy closure has been replaced by the stronger condition of compactness. Their definitions also differ somewhat to reflect the change to continuous measures.

Non-emptiness can be trivially defined as requiring that the program results in at least one measure, the rest are defined formally below.

Up closure allows refinement to be expressed by reverse subset inclusion of the result sets, i.e. program r is refined by r' when $r'.s \subseteq r.s$ (for any initial state s). The formal definition of up-closure is similar to that found in pGCL (Definition 5.24). However, this now uses continuous measures and therefore follows the refinement ordering for continuous probability measures as given in Definition 6.7. To highlight these differences (and to be consistent with the rest of sGCL) μ represents a (continuous) measure in the following definition:

Definition 7.8. A subset $\mathcal{D}$ of $\bar{S}$, a set of measures, is *up closed* if it is closed under refinement of its elements – if for all $\mu, \mu' \in \bar{S}$ then

$$\mu \in \mathcal{D} \text{ and } \mu \sqsubseteq \mu' \text{ implies } \mu' \in \mathcal{D} \ .$$

□

Note that the up closure of abort will be the whole of $\bar{S}$ as all programs refine abort.

Convexity allows a demonic choice of two programs to be refined by any probabilistic choice of the same two programs. Again the formal definition of convexity is close to the one in pGCL (Definition 5.25). However, more clarity is needed as to what is meant by a probabilistic combination of two (continuous) measures. This is formally defined as follows:

Definition 7.9. For two measures $\mu, \mu' \in \bar{S}$, their p-probabilistic combination is defined for set of states $A \in \mathcal{B}_S$ as

$$(\mu \,{}_p\!\oplus\, \mu').A := p * \mu.A + (1-p) * \mu'.A \ .$$

□

Based on this definition of probabilistic choice, convexity can be defined as follows:

Definition 7.10. A set $\mathcal{D}$ of measures is *convex* if for every $\mu, \mu' \in \mathcal{D}$ and probability $p \in [0,1]$ then $\mu \,{}_p\!\oplus\, \mu' \in \mathcal{D}$ also. □

Cauchy closure is related to continuity, which is an important property for showing the existence of fixed points of recursion. For continuous probability, the stronger property of *compactness* is required because of the infinite nature of the state space. Compactness requires the set to be bounded as well as (Cauchy) closed. The definition of Cauchy closure, and hence compactness, is more complex than that found in pGCL (Definition 5.26). It is no longer possible to convert to Euclidean space and require that it is closed in that sense as continuous probability measures have infinite state spaces, and thus infinite axes would be required. Instead measures need to be compared in an alternative metric space. According to van Breugel [82] and Doberkat [22] the most suitable metric space for modelling stochastic non-determinism is the Kantorovich metric space. Recall (Section 5.3) the definition of the Kantorovich metric:

Definition. Given any two Borel probability measures μ and ν on separable metric space (S, d), the *Kantorovich distance* between μ and ν is defined by

$$K(\mu,\nu) := \sup\left\{ \left| \int_S f \mathrm{d}\mu - \int_S f \mathrm{d}\nu \right| \cdot ||f|| \leq 1 \right\} ,$$

where $||\cdot||$ is the *Lipschitz semi-norm* defined by $||f|| = \sup_{x \neq y} \frac{|f.x - f.y|}{d(x,y)}$ for a function $f : S \to \mathbb{R}$. □

Compactness is thus defined for continuous probability according to the Kantorovich metric:

Definition 7.11. A set of measures over S is *compact* if they are compact under the Kantorovich metric. □

Note that Section 5.3 also provides a more intuitive duality of the Kantorovich metric (Definition 5.12) and a simplification for measures over the real numbers (Definition 5.13). These alternative definitions may be used instead when determining compactness where appropriate, but care must be taken when using them with sub-probability (see Appendix D.2). An important result is that the set of all measures defined over a closed interval of reals is compact under the Kantorovich metric [82].

It is now possible to define the model for demonic stochastic programs as follows:

Definition 7.12. Given a state space S, the set of non-empty, up closed, convex and compact subsets of $\bar{S}$ is written $\mathbb{C}S$, and such subsets are said to be *stochastically closed*. It can be shown that $\mathbb{C}S$ is a *cpo* under $\supseteq$.

The *cpo* of demonic probabilistic programs over S is then defined

$$\mathbb{H}S := (S \to \mathbb{C}S, \sqsubseteq) ,$$

where for $r, r' \in \mathbb{H}S$

$$r \sqsubseteq r' := (\forall s : S \cdot r.s \supseteq r'.s) \ .$$

□

The final aspect of the relational semantics for a non-deterministic sGCL is the definition of its operators such as probabilistic choice, demonic choice and sequential composition. In general the definitions are similar to those found in pGCL (Section 5.4.3), but continuous measures are now used.

The probabilistic choice between two non-deterministic sGCL programs is defined as follows:

Definition 7.13. For two programs $r, r' \in \mathbb{H}S$, their p-probabilistic combination is defined as

$$(r \ {}_p\!\oplus\ r').s := \{\mu : r.s; \mu' : r'.s \cdot \mu \ {}_p\!\oplus\ \mu'\} \ .$$

□

Note that this uses the definition of the probabilistic combination of two continuous probability measures as given in Definition 7.9.

The demonic choice is found by taking all possible probabilistic combinations[1] of the programs:

Definition 7.14. For two programs $r, r' \in \mathbb{H}S$, their demonic combination is defined as

$$(r \sqcap r').s := (\cup p : [0,1] \cdot (r \ {}_p\!\oplus\ r').s) \ .$$

□

Sequential composition requires the use of Kleisli composition (Definition 7.7), and is defined as:

Definition 7.15. For two programs $r, r' \in \mathbb{H}S$, their sequential composition is defined as

$$(r; r').s := \{\mu : r.s; f : \mathbb{D}S \mid r' \sqsubseteq f \cdot f^*.\mu\} \ .$$

where $r' \sqsubseteq f$ means $(\forall s : S \cdot f.s \in r'.s)$. □

Therefore the sequential composition is found by taking all the possible intermediate measures μ that r can take from s, and finding for each the Kleisli composition of all the possible deterministic refinements f of r'.

[1] As with pGCL this includes the cases where one of the programs is selected with probability one to allow $r \sqcap r' \sqsubseteq r$, for example.

7.3 Relating the Transformer and Relational Semantics

This section describes the relationship between the transformer and relational semantics of a non-deterministic sGCL. This differs somewhat to that of the deterministic sGCL (as presented in Section 6.4.1) because the Riesz representation theorem no longer applies. Instead the relationship between the semantics is defined in a similar way to that found in pGCL (Section 5.4.4), by providing a relational-to-transformer embedding and a transformer-to-relational retraction. The challenge of showing that the transformer semantics is consistent with the relational model is also discussed.

The relational to transformer *embedding* (translation from the relational model to the transformer model) is defined as follows:

Definition 7.16. The injection $wp \in \mathbb{H}S \to \mathbb{T}S$ is defined

$$wp.r.\beta.s := (\sqcap \mu : r.s \cdot \int_{\mu} \beta) \ ,$$

for program $r \in \mathbb{H}S$, expectation $\beta \in \mathbb{E}S$, and state $s \in S$. □

Informally this states that the relational to transformer embedding of a program selects the measure from the set of measures which satisfies Formula 6.5 and minimises the pre-expectation. Minimum is used due to the demonic interpretation of non-determinism.

Before defining the relationship from the the transformer model to the relational model, it is necessary to defined the set of *regular transformers*, i.e. those that correspond to a valid relational representation:

Definition 7.17. The set of *regular expectations transformers* $\mathbb{T}_r S$ over S is the *wp*-image of $\mathbb{H}S$ in $\mathbb{T}S$, thus defined as

$$\mathbb{T}_r S := \{r : \mathbb{H}S \cdot wp.r\}$$

□

Note that $\mathbb{T}_r S$ is characterised by the set of healthiness conditions described in Section 7.1.1.

The transformer to relational *retraction* (translation from the transformer model to the relational model) is now defined as follows:

Definition 7.18. The function $rp \in \mathbb{T}S \to \mathbb{H}S$ is defined

$$rp.t.s := \{\mu : \bar{S} \mid (\forall \beta : \mathbb{E}S \cdot t.\beta.s \leq \int_{\mu} \beta)\} \ ,$$

for transformer $t \in \mathbb{T}S$ and state $s \in S$. □

7.3.1 Proving the consistency of the two semantics

It is not sufficient to just define the transformer and relational semantics and how they are related. Proving the consistency of the semantics is vital for a model-based specification language. This is particularly challenging when the language combines both continuous probability and demonic non-determinism. This section summarises: the proof effort required; the difficulties of completing these proofs; and the progress made towards the proof goals. A more detailed discussion of the challenges and issues can be found in Appendix D.

The main step required to prove the consistency of the transformer and relational semantics is to show that there is a 1–1 correspondence between $\mathbb{T}_r S$ and $\mathbb{H}S$ through the mutual inverses wp and rp. This involves showing that both wp and rp are injections. It also requires that applying wp to a program in $\mathbb{H}S$ results in a transformer that is both sub-linear and continuous. Another important step is to show that all of the program constructs preserve the relationship between the two semantics. For example, if a probabilistic choice were carried out in the relational semantics, it needs to be shown that the result is equivalent to doing the same probabilistic choice in the equivalent transformer representation of the system. Finally, it needs to be shown that the representation of each program construct in the relational semantics respects the conditions of non-emptiness, up-closure, convexity and compactness. This essentially means that the result of applying a program construct to arguments in $\mathbb{H}S$, must also be in $\mathbb{H}S$.

In order to complete the proofs described above, a significant amount of work is required as they are considerably challenging. In some cases, the proofs require further development of the mathematical theory upon which the semantics is based (such development is beyond the scope of this thesis). In particular, the proofs required to demonstrate the 1–1 correspondence between $\mathbb{T}_r S$ and $\mathbb{H}S$ through the mutual inverses wp and rp are especially challenging. In pGCL, the proofs that wp and rp are injections rely on the use of a representation of discrete distributions in Euclidean space. This allows the re-use of fundamental geometric results. Ideally, the same approach should be taken for sGCL. However, there is no analogous way of representing continuous measures in Euclidean space. An approximation approach for this purpose has been explored (see Appendix D). For the wp injection this approach looks promising. However, the approach results in further challenges relating to compactness that need to be resolved. For example, it is not a trivial problem to translate compactness arguments from the Kantorovich metric space to the approximation representation in Euclidean metric space. These issues are discussed in more detail in Appendix D. The compactness issues also affect the proof that applying wp to a program in $\mathbb{H}S$ results in a transformer that is both sub-linear and continuous. Whilst the remaining proofs have not been explored fully, it is anticipated that these proofs will be less complex as they neither require finite state spaces nor use the conversion to Euclidean space in their pGCL equivalents.

In spite of the challenges described above, significant progress has been made towards showing that wp is an injection. Two approaches to this proof are discussed in Appendix D. The first gets blocked by the problem of converting compactness arguments between the Kantorovich and Euclidean metric spaces. However, the second approach promises a resolution to the problem, although some of the detail still needs to be completed. Some progress has also been made on the rp injection proof. However, the compactness issue encountered in this proof still needs to be resolved. Alternative strategies could involve a different approximation technique for representing continuous measures in Euclidean space or an alternate metric space to the Kantorovich metric space for measures.

Therefore, proving the consistency of a non-deterministic version of sGCL presents a considerable challenge. Significant further research is required to develop the language to the level required of a model-based specification language.

7.4 Refinement Notions

In addition to the refinement notions explored in the deterministic version of sGCL (Sections 6.5.1 and 6.5.2), non-determinism allows a further notion of refinement. A set of measures can be refined by a subset of those measures, thus reducing the amount of non-determinism. This notion of refinement is explored below using an example to illustrate it. Some interesting issues are also discussed.

7.4.1 Reducing non-determinism

Refinement between sets of probability measures is a powerful technique for abstracting and designing software. In the early stages of design the exact measure of a variable may not be known. Instead, a range of possible measures may be considered. An abstract program of the system could model this lack of knowledge using non-determinism between the possible options. Later in the development process when more information is available some of these options may be eliminated from the design. The abstract program could then be refined with a more concrete program that does not include the eliminated options. This notion of refinement is given formally in Definition 7.12.

For technical reasons the sets of measures allowed in sGCL must satisfy certain properties, as was discussed in Section 7.2.2. The property that makes this notion of refinement particularly interesting is convexity. This states that if two measures are in a set of measures, then all possible probabilistic combinations of those measures must also be within it. Thus demonic choice between two measures is defined (Definition 7.14) as the set of probabilistic combinations of those two measures. Whilst this is essential for the mathematics, to ensure that the demonic choice of any two measures is defined, it can be rather counterintuitive.

For example, consider the situation where a component is anticipated to fail accord-

ing to a truncated exponential distribution[2], but the failure rate is not yet known. One might expect

$$x :\oplus exp.1 \sqcap x :\oplus exp.3 \ ,$$

to provide the set of all exponential distributions with rate parameters between one and three. However, what it really means is the set of all *hyperexponential* distributions over the two exponential distributions

$$p * exp.1 + (1 - p) * exp.3 \ ,$$

where p is in $[0, 1]$. What this means for refinement is that

$$x :\oplus exp.1 \sqcap x :\oplus exp.3 \ \not\sqsubseteq \ x :\oplus exp.2$$

because $exp.2$ is not included in the set of measures in the abstract program.

Based on the above one may wish to define the non-determinism over the parameters themselves as follows[3]

$$r :\in [1, 3]; \ x :\oplus exp.r \ .$$

In theory this program would define the set of exponential distributions whose rate parameter is between one and three. It would also include all the possible hyperexponential distributions between these distributions. However, the expression $r :\in [1, 3]$ is an example of infinite demonic choice, which is not allowed in sGCL[4].

A valid (but not as elegant) alternative is to provide a *finite* set of rate parameters between the two possible extremes at some suitable level of granularity. For example, the rate may be chosen from the set $\{1, 1.1, \ldots, 3\}$, which allows any number between one and three to the precision of one decimal place, as shown below.

$$r :\in \{1, 1.1, \ldots, 3\}; \ x :\oplus exp.r \ . \tag{7.1}$$

The result of this now is the set of all exponential distributions with a rate parameter in $\{1, 1.1, \ldots, 3\}$ plus all the possible hyperexponential distributions between these distributions.

The set of valid refinements of the program given in Formula 7.1 include the intuitive

$$x :\oplus exp.1 \sqcap x :\oplus exp.2 \ ,$$

[2]The notation exp.λ is used to represent the exponential distribution with a rate parameter of λ. It is assumed for compactness reasons that the distribution is truncated at some finite value. The details of the truncation are irrelevant for the purposes of this illustration, however.

[3]The notation $x :\in X$ means that x takes any value in the set X. Intervals of real numbers are a special case of this notation.

[4]Recall (Section 7.1) that this is because infinite demonic choice breaks the continuity healthiness condition of sGCL.

$$r :\in \{2, 2.1, \ldots, 3\};\ x :\oplus exp.r\ ,$$

and

$$x :\oplus exp.2\ .$$

However, it also includes less intuitive programs such as

$$x :\oplus 0.3 * exp.1 + 0.7 * exp.2\ ,$$

a hyperexponential distribution between $exp.1$ and $exp.2$.

This is a fascinating trade-off between making the language as generally applicable as possible and providing intuitive refinements of this nature.

7.5 Discussion

This section discusses the language extension to non-determinism explored above and contrasts it with other stochastic formalisms.

The main benefit of including non-determinism in a model-based specification language is to provide a means of abstraction. It allows key properties of a design to be analysed whilst enabling other decisions to be delayed. If the properties of interest hold in spite of the non-determinism included, it allows the implementation to choose between the options at a later date without invalidating those properties. This interpretation of non-determinism is known as demonic – the program is guaranteed to operate correctly in spite of not knowing which direction such decisions will take. The interpretation of non-determinism in sGCL is demonic.

An alternative interpretation of non-determinism is *angelic*. This assumes that the option that gives the best outcome, with respect to the properties of interest, will always be chosen. This alternative is not explored in sGCL, however McIver and Morgan discuss this option for pGCL [61, Section 8.5].

As far as the author is aware, this is the first attempt to combine demonic non-determinism and arbitrary continuous probability distributions in a formal language. Continuous Time Markov Decision Processes (CTMDPs), which essentially add the option of non-determinism to CTMCs for the purposes of optimisation (see Section 2.1.4), are supported by model checkers such as MRMC [51, 52]. However, as with CTMCs, the continuous probability in CTMDPs is limited to the exponential distribution.

The semantics presented here are mainly based on those found in McIver and Morgan's pGCL [61]. However, pGCL only covers discrete probability. Therefore, significant adaptation of the pGCL semantics was required, particularly for the relational semantics. Parts of the relational semantics were inspired by: Giry's stochastic powerdomain [31], in particular the Kleisli composition; and van Breugel's use of the Kantorovich met-

ric for analysing the compactness of sets of measures [82]. Neither Giry nor van Breugel present a proof theory to complement their semantics. Further, Giry does not consider non-determinism in his semantics either. Finally, there are once more similarities with Kozen's early work on developing probabilistic formalisms [53, 54]. However, his research considers probability to be a replacement for non-determinism, not an addition to it.

Demonstrating the consistency of the transformer and relational semantics of sGCL proved to be highly challenging. As such there are still unresolved issues on how to complete this proof work. This constitutes a significant area for further research. Although no contradictions to the consistency proofs have been found, the non-deterministic version of sGCL does not satisfy the requirements for a model-based specification language until these proofs have been completed. Therefore the case study explored in the following chapter focuses on the deterministic features of sGCL as defined and examined in Chapter 6. Even when these challenges are resolved the non-deterministic version of sGCL (as per the deterministic version) is limited to distributions over finite intervals, and recursion is limited to finite while-loops.

Part III

Applying and Evaluating sGCL

Chapter 8

Applying sGCL: Patterns and Pitch

This chapter demonstrates the practical application of sGCL in a two-part case study. The first part of the case study develops and examines a new design pattern that addresses the problem of using multiple, drift-prone sensors to obtain a required value. The fact that these sensors can drift necessitates additional monitoring to traditional voting techniques. The second part of the case study explores the use of this design pattern in the analysis of a pitch monitor in an aeroplane, where sensors are known to drift. Recall (Section 6.6) that the flash filestore case study is not revisited here, due to the state space restrictions of sGCL.

Both parts of the case study illustrate the power of sGCL in analysing the interaction between multiple continuous probability distributions. The analysis of such interaction is much more elegant and powerful in sGCL than it would be in a discrete approximation of the situation. The opportunities for sGCL refinements in the case study are also discussed.

The notion of design patterns is introduced in Section 8.1, along with a description of the design pattern for a monitoring voter. An sGCL model of a monitoring voter is given and analysed in Section 8.2. The pitch monitor case study is introduced and analysed in sGCL in Section 8.3. Finally further opportunities for sGCL modelling and analysis relating to the monitoring voter pattern are discussed in Section 8.4.

8.1 Design Patterns

In software engineering it is desirable to reuse existing designs where possible to reduce complexity. The use of design patterns [29] is a well established method for recording and enabling the reuse of design elements that commonly occur. The idea behind design patterns is a simple one born out of architecture and the common patterns that occur in buildings. The essence of a design pattern describes a solution to a common

design problem along with possible consequences or required trade-offs associated with its application.

Patterns describing faults and fault tolerant architectures are of particular interest because the goal of this thesis is to enable the rigorous analysis and comparison of approaches for improving dependability. A library of design patterns to describe common faults and fault tolerance patterns in embedded systems is being developed by the DESTECS European project[1] [13, 70]. Based on the structure of software engineering patterns [29], the following fields may be used to describe fault tolerance design patterns:

- **Name** – A meaningful identifier for the pattern.
- **Intent** – A short statement about what problem the pattern is intended to address and what it does.
- **Also known as** – Other well-known names for the pattern.
- **Motivation** – A scenario that illustrates the use of the pattern in solving the identified design problem.
- **Applicability** – What are the situations in which the pattern can be applied and how can these situations be recognised?
- **Structure** – A graphical representation of the structural aspects of the pattern, for example, a UML class diagram.
- **Consequences** – The trade-offs and results of using the pattern.
- **Known uses** – Examples of the patterns found in real systems.
- **Related patterns** – Which other patterns are closely related and how do they differ?
- **Fault assumptions** – The fault modes that the pattern is designed to handle.

The DESTECS pattern library includes fault modes such as noise, drift and bit flips. Noise describes the situation where random errors occur in a sensor reading (usually in the conversion process from analogue to digital). Drift describes a more systematic error, where sensor readings diverge from the values they measure over time. In a bit flip fault the sensor reading may provide a wildly inaccurate value (as opposed to noise where the error is usually fairly small). The DESTECS pattern library also includes elements for fault tolerance such as the voter and monitor patterns. The voter pattern describes the process of combining the inputs from several sensors with the aim of providing a more reliable reading. The monitor pattern describes the process of monitoring a system

[1] The EU FP7 project on Design Support and Tooling for Embedded Control Software (Grant Agreement Number: INFSO-ICT-248134), www.destecs.org.

controller and intervening when it tries to perform an unsafe action. A more detailed description of the voter and monitor patterns can be found in Appendix F.

The scenario in Section 8.3 requires a combination of the features described in the voter and monitor patterns. In this situation voting is used to provide a result, but the sensors (that provide the inputs on which voting takes place) are prone to drift. Therefore the outputs of the sensors are monitored and if they drift too far apart a recovery action is taken to reset the voters. The recovery action is considered external to the monitoring voter as this often requires external intervention (e.g. a pilot levelling off to reset the attitude sensors, see Section 8.3). A new "monitoring voter" pattern is described to capture this design. The design pattern for the *monitoring voter* is detailed below:

Name

Monitoring voter.

Intent

To produce a single sensor reading from multiple (redundant or diverse) sensor inputs. Indicates that a reset action is required when the sensor readings diverge with respect to a given tolerance and reset logic.

Also known as

N/A.

Motivation

Multiple sensor readings are taken to allow for potential failure, noise or drift in sensor readings. The sensors may be replicated (multiple copies of the same sensor) or diverse (each sensor has a different design). The voter provides a strategy for combining these readings aimed at increasing the dependability of the reading. Example strategies include taking the median result or having a majority vote. The monitoring extension to the voter pattern applies in the situation where the sensors can drift over time. A monitor is added to the pattern to provide a strategy for deciding when to trigger a reset action, for example, when two or more sensors differ from each other by more than an allowed tolerance.

Applicability

Use the monitoring voter pattern when:

- an accurate sensor reading is required;
- the sensors available are prone to (measurement) errors;

- the sensors available are prone to drift;
- and a technique for resetting (or recalibrating) the sensors is available[2].

Structure

The monitoring voter consists of several components, which are shown diagrammatically below. The pattern requires two or more sensors, a voter and a monitor. The monitor tests the level of drift of the sensors. If this is not acceptable, the need for a reset action is alerted. The voter combines the results from the sensors. The strategy design pattern [29] can be used to encapsulate the voting and monitoring algorithms. The strategy pattern essentially encapsulates a family of algorithms and makes them interchangeable. This would allow different voting and monitoring strategies to be seamlessly explored.

This structure is shown diagrammatically through a UML class and object diagram in Figure 8.1. The object diagram shows that the monitoring voter holds a reference to a voter and a monitor object, which in turn have references to the same set of sensors, for the purposes of voting and monitoring respectively.

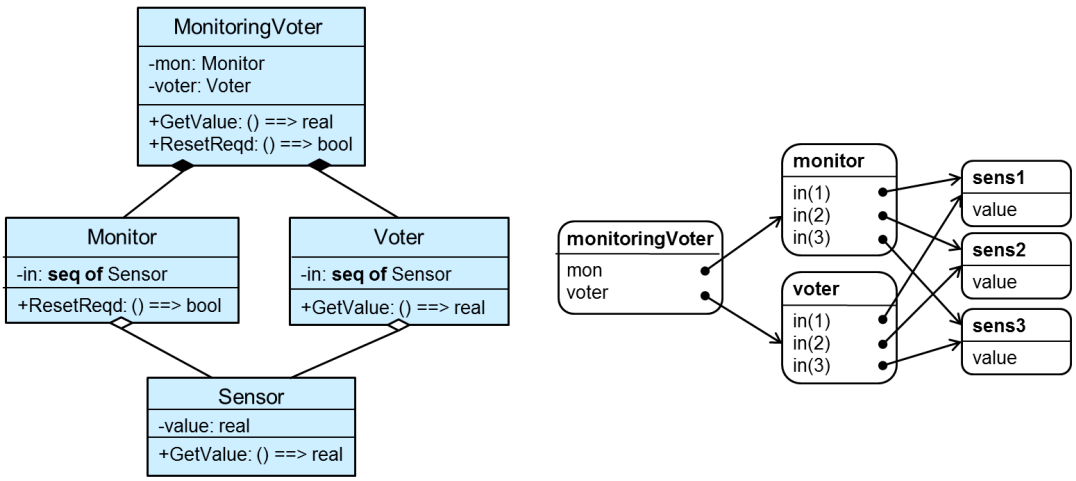

Figure 8.1: Class (left) and object (right) diagrams for the monitoring voter pattern

Consequences

The benefit of voting is that a given number of concurrent failures can be tolerated in a timely manner (without requiring any rollback). However, it is well known that the expected benefits are often not fully achieved because of common mode failures, caused by insufficient diversity in the sensors [60]. The monitoring voter adds the benefit of recovering from adverse drifting of the sensors. However, this is not without cost as it may require external intervention.

[2]This may involve some external interaction, such as draining a tank, to achieve.

Known uses

The monitoring voter pattern is found in the aerospace industry in monitoring the pitch of an aircraft (see Section 8.3).

Related patterns

This pattern is a combination of the voter and monitor patterns from DESTECS pattern library [13]. The pattern extends the voter pattern by adding drift monitoring to it. The structure pattern [29] could also be used to encapsulate voting and monitoring algorithms.

Fault assumptions

This pattern is primarily designed for sensors with (measurement) errors and a fault mode of drift. The voting aspect of the pattern provides tolerance of errors, and for n sensors the monitoring logic may provide detection and recovery of independent drift in up to n sensors (if all n sensors drift at the same rate and in the same direction this cannot be detected).

In addition voting may provide (for n sensors) tolerance of up to $n - 2$ sequential faults (assuming that a sensor is retired once identified as faulty) [58], and detection of independent (concurrent) faults. These faults are assumed to be value (fault modes that affect the value of a sensor reading such as bit flips, noise or stuck at x) or omission[3] faults.

There is already extensive research on whether voting is a suitable alternative to a single sensor strategy in terms of both the functionality and the reliability benefits it provides. For example, Iliasov et al. [42] demonstrate that n-version programming[4] is a valid refinement pattern in Event-B, whilst Laprie et al. [58] analyse the reliability gains achieved by n-version programming. Therefore the rest of this chapter focuses on the monitoring aspects of the monitoring voter, and the probability of requiring a reset action under various circumstances. This aspect can be nicely modelled and analysed using sGCL.

8.2 The Monitoring Voter in sGCL

In this section the reset logic aspect of the monitoring voter is explored. This logic determines when a reset of the sensors is required. An sGCL model is created of a simple monitoring voter scenario and the probability of requiring a reset action is calculated for

[3] Under the assumption that an omission fault results in an inaccurate (old) value (or detectable error code) to be communicated for that sensor reading.

[4] N-version programming [8] is a technique where n versions of a program are created and voting is used to determine the overall result.

$x \quad :\oplus \quad U.[0,1];$
$y \quad :\oplus \quad U.[0,1];$

if $|x - y| > T$ then
  abort
else
  skip
fi

– x, y represent the error in the reading on sensors x and y respectively
– T represents the maximum difference tolerated between the two sensor readings
– $|i|$ represents the absolute value of variable i

Figure 8.2: A monitoring voter with two replicated sensors

various sets of sensors. This forms the basis for analysing the aeroplane pitch monitor in Section 8.3.

A simple monitoring voter consists of two sensors, a voter and a monitor. The logic for the monitor in this case is simply to trigger a reset action when the two sensors provide readings that are different by more than some given tolerance value. A reset action is an undesirable and costly procedure, therefore the aim is to analyse and minimise the occurrence of it happening. The sGCL model abstracts away from the timing issues and only considers a single request for data from the sensors and whether or not that would trigger a reset action. Each sensor has an associated (continuous) probability distribution for the error it can provide on the reading. For simplicity, the uniform distribution is used in the analysis that follows, although sensors following any bounded[5] continuous probability distribution could be analysed in a similar manner.

The first example to be considered has two replicated[6] sensors that have a random error following the uniform distribution on $[0, 1]$ (written $U.[0, 1]$). This is modelled in sGCL as shown in Figure 8.2. The situation where a reset action is required is modelled as the program which aborts, as this is the bad result that needs to be minimised. The real value being measured is omitted from the model as it does not affect the difference between the two readings.

To analyse this model in sGCL the post-expectation of interest is simply $[true]$, to give the probability that a reset action is not triggered. The formal proof follows below:

$$
\begin{aligned}
& wp.prog.[true] \\
\equiv \quad & \qquad \text{definition of } prog \\
& wp.\,(x :\oplus U.[0,1]; y :\oplus U.[0,1]; \mathsf{if}\ |x-y| > T\ \mathsf{then\ abort\ else\ skip\ fi})\,.[true]
\end{aligned}
$$

[5]The possible values that a random variable of the distribution could take are bounded above and below by some finite number.

[6]Note that these sensors are replicated in the sense that they have the same error distribution, but are assumed to have independent observations from that distribution at all times.

$\equiv$ sequential composition

$$wp.\,(x :\oplus U.[0,1])\,.(wp.\,(y :\oplus U.[0,1])\,.\\ \quad (wp.\,(\mathsf{if}\ |x-y|>T\ \mathsf{then\ abort\ else\ skip\ fi})\,.[true]))$$

$\equiv$ conditional

$$wp.\,(x :\oplus U.[0,1])\,.(wp.\,(y :\oplus U.[0,1])\,.\\ \quad ([|x-y|>T]*wp.\mathsf{abort}.[true]+[\neg\,(|x-y|>T)]*wp.\mathsf{skip}.[true]))$$

$\equiv$ abortion, identity and definition of *true*

$$wp.\,(x :\oplus U.[0,1])\,.(wp.\,(y :\oplus U.[0,1])\,.\\ \quad ([|x-y|>T]*0+[\neg\,(|x-y|>T)]*1))$$

$\equiv$ simple algebra

$$wp.\,(x :\oplus U.[0,1])\,.\,(wp.\,(y :\oplus U.[0,1])\,.[|x-y|\leq T])$$

$\equiv$ stochastic assignment

$$wp.\,(x :\oplus U.[0,1])\,.\left(\int_{[|x-y|\leq T]}\mathrm{d}U.[0,1]\right)$$

$\equiv$ integration theory

$$wp.\,(x :\oplus U.[0,1])\,.\left(\begin{array}{lll} & [x<T]*\displaystyle\int_0^{x+T}1\ \mathrm{d}y \\ +\ & [T\leq x\leq 1-T]*\displaystyle\int_{x-T}^{x+T}1\ \mathrm{d}y \\ +\ & [x>1-T]*\displaystyle\int_{x-T}^{1}1\ \mathrm{d}y \end{array}\right)$$

$\equiv$ integration theory and simple algebra

$$wp.\,(x :\oplus U.[0,1])\,.\left(\begin{array}{lll} & [x<T]*(x+T) \\ +\ & [T\leq x\leq 1-T]*2T \\ +\ & [x>1-T]*(1+T-x) \end{array}\right)$$

$\equiv$ stochastic assignment

$$\int_{[x<T]*(x+T)+[T\leq x\leq 1-T]*2T+[x>1-T]*(1+T-x)}\mathrm{d}U.[0,1]$$

$\equiv$ integration theory

$$\int_0^T 1*(x+T)\ \mathrm{d}x+\int_T^{1-T}1*2T\ \mathrm{d}x+\int_{1-T}^{1}1*(1+T-x)\ \mathrm{d}x$$

$\equiv$ integration theory

$$\left(\tfrac{T^2}{2}+T^2\right)+\left(2T\,(1-T)-2T^2\right)+\left(1+T-\tfrac{1}{2}-(1+T)\,(1-T)+\tfrac{(1-T)^2}{2}\right)$$

$\equiv$ $2T-T^2$ simple algebra

The probability of the two sensors triggering a reset is $1-2T+T^2$ or $(1-T)^2$. It is pretty clearly to see that the probability of a reset action decreases as the tolerance value T increases and vice versa. This is fairly intuitive, but it provides confirmation

$$
\begin{array}{lll}
x & :\oplus & U.[-a,a]; \\
y & :\oplus & U.[-b,b]; \\
 & & \\
\textsf{if} & |x-y| > T & \textsf{then} \\
 & \textsf{abort} & \\
\textsf{else} & & \\
 & \textsf{skip} & \\
\textsf{fi} & &
\end{array}
$$

- x, y represent the error in the reading on sensors x and y respectively
- a, b represent the maximum errors in the reading on sensors x and y respectively
- T represents the maximum difference tolerated between the two sensor readings
- $|i|$ represents the absolute value of variable i

Figure 8.3: A monitoring voter with two diverse sensors

that the result of the analysis is reasonable.

The second example investigates the probability of a reset action for two diverse sensors. Each sensor has a different range of possible errors on a given reading. For simplicity, it is assumed that each sensor is as likely to underestimate a reading as overestimate. Again the uniform distribution will be used to simplify the calculation, but the approach is applicable to any bounded continuous probability distribution. The sGCL model for the diverse sensors is given in Figure 8.3.

The sGCL model of the monitoring voter with two diverse sensors can also be analysed in sGCL using the post-expectation $[true]$. However, it is a little more complex because the result depends on the relationship between the maximum errors of the sensor readings (a and b) and the tolerance T. The proof follows below:

$$
\begin{array}{ll}
 & wp.prog.[true] \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } prog \\
 & wp.\left(x :\oplus U.[-a,a]; y :\oplus U.[-b,b]; \textsf{if } |x-y| > T \textsf{ then abort else skip fi}\right).[true] \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sequential composition} \\
 & wp.\left(x :\oplus U.[-a,a]\right).\left(wp.\left(y :\oplus U.[-b,b]\right).\right. \\
 & \qquad \left.\left(wp.\left(\textsf{if } |x-y| > T \textsf{ then abort else skip fi}\right).[true]\right)\right) \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{conditional} \\
 & wp.\left(x :\oplus U.[-a,a]\right).\left(wp.\left(y :\oplus U.[-b,b]\right).\right. \\
 & \qquad \left.\left([|x-y| > T] * wp.\textsf{abort}.[true] + [\neg\left(|x-y| > T\right)] * wp.\textsf{skip}.[true]\right)\right) \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{abortion, identity and definition of } true \\
 & wp.\left(x :\oplus U.[-a,a]\right).\left(wp.\left(y :\oplus U.[-b,b]\right).\right. \\
 & \qquad \left.\left([|x-y| > T] * 0 + [\neg\left(|x-y| > T\right)] * 1\right)\right)
\end{array}
$$

$\equiv$ simple algebra

$$wp.\,(x :\oplus U.[-a,a])\,.\,(wp.\,(y :\oplus U.[-b,b])\,.[|x-y| \leq T])$$

$\equiv$ stochastic assignment

$$wp.\,(x :\oplus U.[-a,a])\,.\left(\int_{[|x-y|\leq T]} \mathrm{d}U.[-b,b]\right)$$

$\equiv$ integration theory

$$wp.\,(x :\oplus U.[-a,a])\,.$$
$$\left(\begin{array}{ll} & [a < b-T] * \left(\; [-a \leq x \leq a] * \displaystyle\int_{x-T}^{x+T} \frac{1}{2b}\; \mathrm{d}y \; \right) \\ + & [b-T \leq a \leq b+T] * \left(\begin{array}{ll} & [-a \leq x < -b+T] * \displaystyle\int_{-b}^{x+T} \frac{1}{2b}\; \mathrm{d}y \\ + & [-b+T \leq x \leq b-T] * \displaystyle\int_{x-T}^{x+T} \frac{1}{2b}\; \mathrm{d}y \\ + & [b-T < x \leq a] * \displaystyle\int_{x-T}^{b} \frac{1}{2b}\; \mathrm{d}y \end{array} \right) \\ + & [a > b+T] * \left(\begin{array}{ll} & [-b-T \leq x < -b+T] * \displaystyle\int_{-b}^{x+T} \frac{1}{2b}\; \mathrm{d}y \\ + & [-b+T \leq x \leq b-T] * \displaystyle\int_{x-T}^{x+T} \frac{1}{2b}\; \mathrm{d}y \\ + & [b-T < x \leq b+T] * \displaystyle\int_{x-T}^{b} \frac{1}{2b}\; \mathrm{d}y \end{array} \right) \end{array}\right)$$

$\equiv$ integration theory and simple algebra (define β as this post-expectation)

$$wp.\,(x :\oplus U.[-a,a])\,.$$
$$\left(\begin{array}{ll} & [a < b-T] * \frac{1}{2b} * \left(\; [-a \leq x \leq a] * 2T \; \right) \\ + & [b-T \leq a \leq b+T] * \frac{1}{2b} * \left(\begin{array}{ll} & [-a \leq x < -b+T] * (x+b+T) \\ + & [-b+T \leq x \leq b-T] * 2T \\ + & [b-T < x \leq a] * (b+T-x) \end{array} \right) \\ + & [a > b+T] * \frac{1}{2b} * \left(\begin{array}{ll} & [-b-T \leq x < -b+T] * (x+b+T) \\ + & [-b+T \leq x \leq b-T] * 2T \\ + & [b-T < x \leq b+T] * (b+T-x) \end{array} \right) \end{array}\right)$$

$\equiv$ $\displaystyle\int_{\beta} \mathrm{d}U.[-a,a]$ stochastic assignment and definition of β above

$\equiv$ integration theory and definition of β

$$
\begin{array}{lll}
 & [a < b - T] * \frac{1}{2b} * \left(\int_{-a}^{a} \frac{1}{2a} * 2T \ \mathrm{d}x \right) & \\
+ & [b - T \leq a \leq b + T] * \frac{1}{2b} * \left(\begin{array}{ll} & \int_{-a}^{-b+T} \frac{1}{2a} * (x + b + T) \ \mathrm{d}x \\ + & \int_{-b+T}^{b-T} \frac{1}{2a} * 2T \ \mathrm{d}x \\ + & \int_{b-T}^{a} \frac{1}{2a} * (b + T - x) \ \mathrm{d}x \end{array} \right) & \\
+ & [a > b + T] * \frac{1}{2b} * \left(\begin{array}{ll} & \int_{-b-T}^{-b+T} \frac{1}{2a} * (x + b + T) \ \mathrm{d}x \\ + & \int_{-b+T}^{b-T} \frac{1}{2a} * 2T \ \mathrm{d}x \\ + & \int_{b-T}^{b+T} \frac{1}{2a} * (b + T - x) \ \mathrm{d}x \end{array} \right) &
\end{array}
$$

$\equiv$ integration theory

$$
\begin{array}{lll}
 & [a < b - T] * \frac{1}{2b} * \frac{1}{2a} * 2T * 2a & \\
+ & [b - T \leq a \leq b + T] * \frac{1}{2b} * \frac{1}{2a} * \left(\begin{array}{ll} & \frac{(-b+T)^2}{2} + (-b + T)(b + T) - \frac{a^2}{2} \\ & + a(b + T) \\ + & 2T(2b - 2T) \\ + & a(b + T) - \frac{a^2}{2} - (b - T)(b + T) \\ & + \frac{(b-T)^2}{2} \end{array} \right) & \\
+ & [a > b + T] * \frac{1}{2b} * \frac{1}{2a} * \left(\begin{array}{ll} & \frac{(-b+T)^2}{2} + (-b + T)(b + T) - \frac{(-b-T)^2}{2} \\ & - (-b - T)(b + T) \\ + & 2T(2b - 2T) \\ + & (b + T)^2 - \frac{(b+T)^2}{2} - (b - T)(b + T) + \frac{(b-T)^2}{2} \end{array} \right) &
\end{array}
$$

$\equiv$ algebra

$$
\begin{array}{lll}
 & [a < b - T] * \frac{T}{b} & \\
+ & [b - T \leq a \leq b + T] * \frac{1}{4ab} * \left(-a^2 + 2a(b + T) - (b - T)^2 \right) & \\
+ & [a > b + T] * \frac{T}{a} &
\end{array}
$$

The probability of the two sensors triggering a reset is $1 - \frac{T}{b}$ when $a < b - T$, $\frac{1}{4ab} * (a + b - T)^2$ when $b - T \leq a \leq b + T$, and $1 - \frac{T}{a}$ when $a > b + T$. Again, the probability of a reset action decreases as the tolerance value T increases and vice versa. Notice that in this second example, the value being integrated w.r.t. x is $\frac{1}{2a}$, and w.r.t y is $\frac{1}{2b}$, instead of one as in the first example. These come from the probability density functions (pdf) of the uniform distributions, $U.[-a, a]$ and $U.[-b, b]$, respectively. If the sensors were to follow different distributions, these values would be replaced by the pdf of the required distribution.

The probabilities of triggering a reset were analysed for a variety of pairs of sensors (all following uniform distributions). The results of these analyses are given in Figure 8.4.

Sensor x	Sensor y	Reset Probability
$U.[0,1]$	$U.[0,1]$	$(1-T)^2$
$U.[-1,1]$	$U.[-1,1]$	$\frac{1}{4}(2-T)^2$
$U.[0,a]$	$U.[0,a]$	$\frac{1}{a^2}(a-T)^2$
$U.[-a,a]$	$U.[-a,a]$	$\frac{1}{4a^2}(2a-T)^2$
$U.[0,a]$	$U.[0,b]$	$[a<b]*\left(\begin{array}{ll} & [a<T]*\frac{1}{2ab}(b-T)^2 \\ + & [b-a<T<a]*\frac{1}{2ab}\left((a-T)^2+(b-T)^2\right) \\ + & [T<b-a]*\frac{1}{2ab}(T(T-4a)+2ab)\end{array}\right)$
		$+\ [b<a]*\left(\begin{array}{ll} & [b<T]*\frac{1}{2ab}(a-T)^2 \\ + & [a-b<T<b]*\frac{1}{2ab}\left((a-T)^2+(b-T)^2\right) \\ + & [T<a-b]*\frac{1}{2ab}(T(T-4b)+2ab)\end{array}\right)$
$U.[-a,a]$	$U.[-b,b]$	$[a<b-T]*\left(1-\frac{T}{b}\right)$
		$+\ [b-T\leq a\leq b+T]*\frac{1}{4ab}*(a+b-T)^2$
		$+\ [a>b+T]*\left(1-\frac{T}{a}\right)$

The notation $U.[a,b]$ stands for the continuous uniform distribution over the interval $[a,b]$. The reset probability in the last two cases depends on the relationship between a, b, and T. In these cases read $[pred]*prob$ to mean the reset probability is equal to $prob$ when $pred$ holds.

Figure 8.4: The probability of triggering a reset action for a variety of sensor pairs

To illustrate these results the reset probability has been plotted against the tolerance level for a variety of these pairs (see Figure 8.5).

8.3 The Aeroplane Pitch Monitor

This section describes and analyses an application of the monitoring voter, that of a pitch monitor in an aeroplane [17]. The pitch of an aeroplane is its degree of tilt about a lateral axis (see Figure 8.6). In order to get an acceptable pitch measurement, diverse sensors must be used and compared as per the monitoring voter pattern. The scenario is described in more detail in Section 8.3.1. The pitch monitor is then formally modelled and analysed in sGCL (Section 8.3.2), building on the abstract analysis of the monitoring voter in Section 8.2.

8.3.1 Overview of the pitch monitor

The orientation of an aeroplane in three-dimensional space is split into three different measurements (or *attitudes*). These are the pitch, roll and yaw, as illustrated[7] in Figure 8.6. Coombes et al. [17] discuss the need to include a "gust alleviation filter"[8] within the flight control system of an aircraft in order to reduce the impact of cross-winds. In order for the filter to function correctly it requires measurements of the pitch and roll

[7]The aeroplane attitude diagram was taken from Coombes et al. [17].

[8]When cross-winds are detected the gust alleviation filter activates special gust-damping flaps.

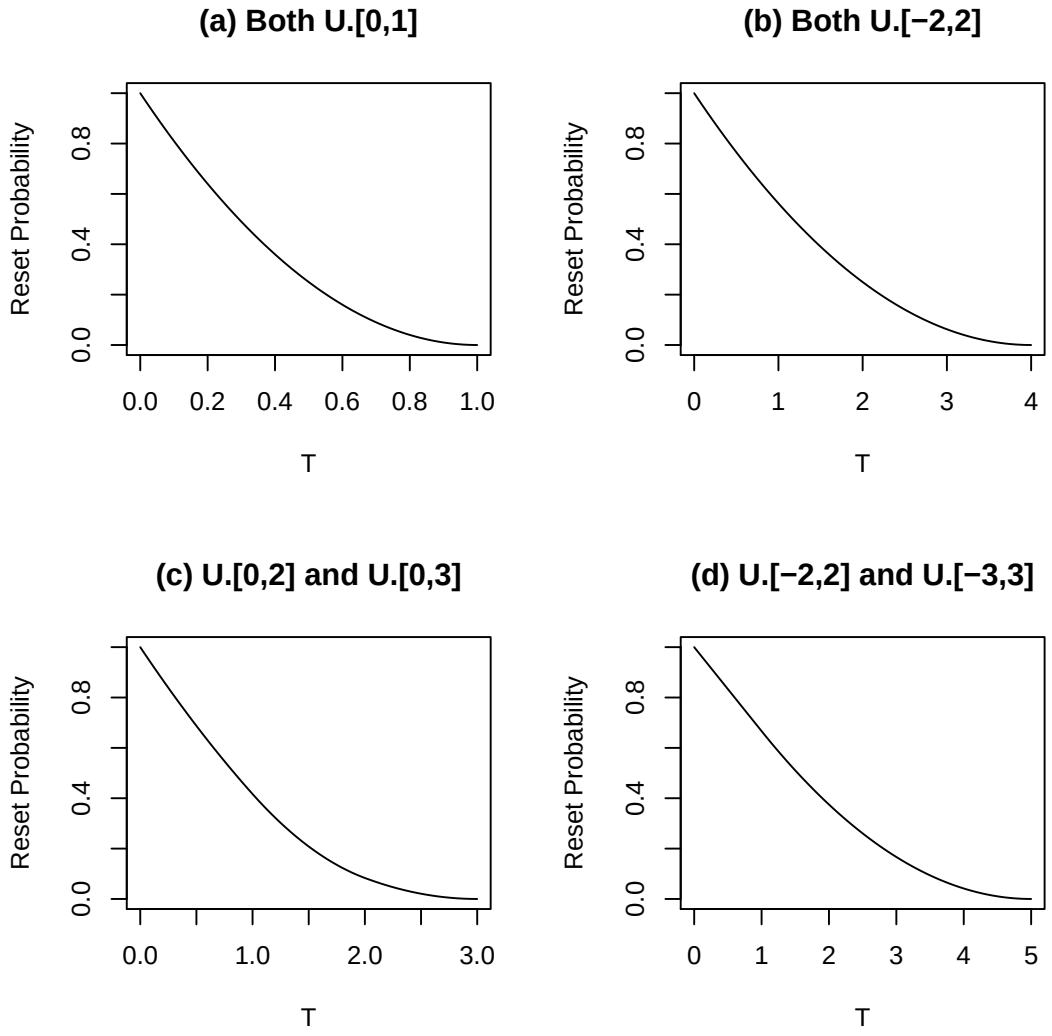

Figure 8.5: Probability of a reset action against tolerance for various sensors

of the aeroplane. Whilst the filter itself is not safety critical, because it is placed within the flight control system it has to be treated as a safety critical component. This means that the pitch and roll measurements that it acts on have to be "high integrity"[9]. This study focuses on the pitch measurements, although a similar discussion could be applied to the roll measurements.

There are a number of components in the aeroplane that are capable of providing pitch measurements. One source of pitch data is the "air motion sensor units" (AMSUs), however these are not considered to be high integrity. A second potential source of pitch data is to derive it (by integration) from the body rate data (rate of change of pitch, roll and yaw). Unfortunately, whilst this measurement is considered to be of sufficient integrity over short periods of time, it is prone to drift over the duration of a flight.

[9]The exact criteria used to decide whether a measurement is "high integrity" or not is unknown.

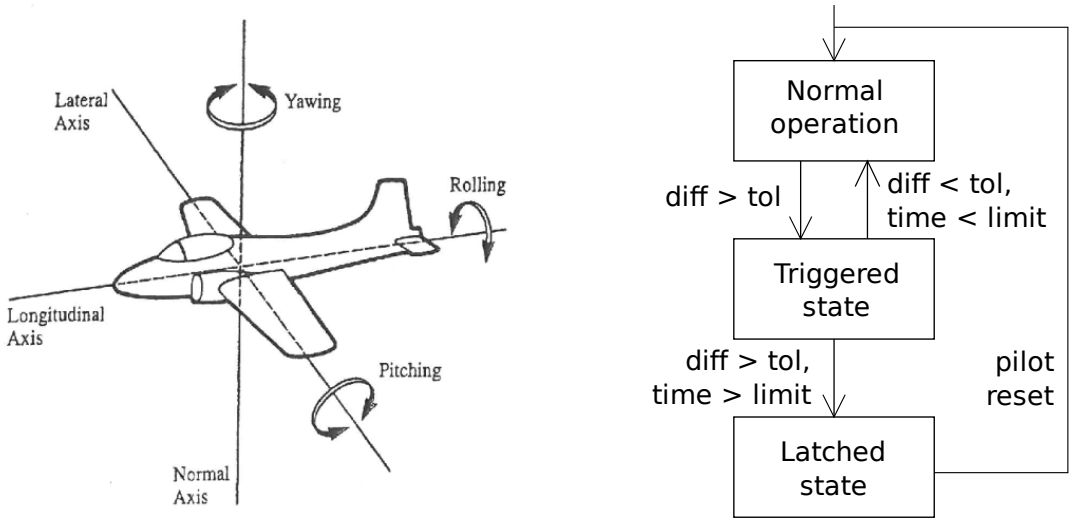

Figure 8.6: Aeroplane attitudes and axes (left) and pitch monitor state chart (right)

Finally the "inertial navigation unit" (IN) also provides a low integrity (independent) source of pitch data.

The chosen solution [17] was to use the pitch data from the AMSUs and check it against the data from the IN. If these two measurements were found to differ by more than a given tolerance the system would enter a triggered state. If the measurements remained out of tolerance of each other for a set time period the system would enter a latched state, otherwise it would revert back to the normal state (see Figure 8.6). Once the system was in the latched state the body rate data was used to provide the pitch data until the pilot performed a manual reset of the AMSUs and IN. This required the pilot to make the aeroplane level first, and is therefore not a desirable state.

The case study focuses on analysing the probability of the system entering the latched state and requiring a manual reset. Given a number of safe options for the sensors and tolerance level, it is desirable to choose that which causes the least inconvenience to the pilot.

8.3.2 Modelling and analysing the pitch monitor in sGCL

The sGCL model of the pitch monitor focuses on the measurement errors of the sensors and the resulting state transfers to the triggered and latched states. The timing aspects are abstracted away into two snapshots of the system state. The system is assumed to enter the triggered state if the first pair of measurements differ by the given tolerance. If the subsequent pair of measurements also differ by the given tolerance the system is assumed to enter the latched state, which is modelled by `abort` as it is considered an undesirable event. Figure 8.7 shows the sGCL model of the pitch monitor. The in-line comments explain how this abstract model links back to the pitch monitor scenario.

Notice the similarity between this model and the sGCL model of the monitoring voter. It is essentially two instances of the monitoring voter pattern nested within

```
x :⊕ U.[−a, a];          % Sensor error for the first AMSU reading
y :⊕ U.[−b, b];          % Sensor error for the first IN reading

if |x − y| > T then      % First readings are out of tolerance, enter triggered state
   x :⊕ U.[−c, c];       % Sensor error for the second AMSU reading
   y :⊕ U.[−d, d];       % Sensor error for the second IN reading

   if |x − y| > T then   % Second readings are out of tolerance
      abort              % Enter latched state
   else
      skip               % Revert to normal state
   fi

else
   skip                  % Stay in normal state
fi
```

- x represents the error in the pitch reading for the AMSU
- y represents the error in the pitch reading for the IN
- a represents the maximum error for the AMSU in the normal state
- b represents the maximum error for the IN in the normal state
- c represents the maximum error for the AMSU in the triggered state
- d represents the maximum error for the IN in the triggered state
- T represents the maximum difference tolerated between the AMSU and IN readings

Figure 8.7: Specification of the pitch monitor

each other. This allows the re-use of some of the analysis from the previous section when finding the probability of entering the latched state. Also note that the error distributions of the second readings differ from the first. This is to model deterioration in the accuracy of the sensors. Without this deterioration there would be no benefit in carrying out a reset action.

As before the sGCL model is analysed by calculating the probability of the model successfully terminating, using the post-expectation $[true]$. However, in this case the core of the analysis has already been completed in the analysis of the monitoring voter pattern. This analysis can be re-used for the pitch monitor. To enhance the readability of the analysis, the following definitions are used:

$$p := wp.\,(x :\oplus U.[-a, a])\,.\,(wp.\,(y :\oplus U.[-b, b])\,.\,([|x - y| > T)])\ ,$$

and

$$q := wp.\,(x :\oplus U.[-c, c])\,.\,(wp.\,(y :\oplus U.[-d, d])\,.\,([|x - y| > T)])\ .$$

Based on the analysis of the monitoring voter in Section 8.2, the values of these expressions are known to be equivalent to

$$\begin{array}{rcl} p & \equiv & [a < b - T] * \left(1 - \frac{T}{b}\right) \\ & + & [b - T \leq a \leq b + T] * \frac{1}{4ab} * (a + b - T)^2 \\ & + & [a > b + T] * \left(1 - \frac{T}{a}\right) \ , \end{array}$$

and

$$\begin{array}{rcl} q & \equiv & [c < d - T] * \left(1 - \frac{T}{d}\right) \\ & + & [d - T \leq c \leq d + T] * \frac{1}{4cd} * (c + d - T)^2 \\ & + & [c > d + T] * \left(1 - \frac{T}{c}\right) \ , \end{array}$$

respectively. The analysis of the pitch monitor then proceeds as follows:

$wp.prog.[true]$

$\equiv$ definition of *prog*

$wp.(x :\oplus U.[-a,a]; y :\oplus U.[-b,b];$ if $|x-y| > T$ then
$(x :\oplus U.[-c,c]; y :\oplus U.[-d,d];$ if $|x-y| > T$ then abort else skip fi)
else skip fi$).[true]$

$\equiv$ sequential composition

$wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,.$
$(wp.($if $|x-y| > T$ then
$(x :\oplus U.[-c,c]; y :\oplus U.[-d,d];$ if $|x-y| > T$ then abort else skip fi)
else skip fi$).[true]))$

$\equiv$ conditional

$wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,.$
$([|x-y| > T] * wp.(x :\oplus U.[-c,c]; y :\oplus U.[-d,d];$
if $|x-y| > T$ then abort else skip fi$).[true]$
$+ [\neg\,(|x-y| > T)] * wp.$skip$.[true]))$

$\equiv$ sequential composition, identity and definition of *true*

$wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,.$
$([|x-y| > T] * wp.\,(x :\oplus U.[-c,c])\,.(wp.\,(y :\oplus U.[-d,d])\,.$
$(wp.\,($if $|x-y| > T$ then abort else skip fi$)\,.[true]))$
$+ [\neg\,(|x-y| > T)] * 1))$

$\equiv$ conditional

$wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,.$
$([|x-y| > T] * wp.\,(x :\oplus U.[-c,c])\,.(wp.\,(y :\oplus U.[-d,d])\,.$
$([|x-y| > T] * wp.$abort$.[true] + [\neg\,(|x-y| > T)] * wp.$skip$.[true]))$
$+ [\neg\,(|x-y| > T)]))$

$\equiv$ abortion, identity and definition of *true*

$$
\begin{array}{l}
wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,. \\
\quad ([|x-y|>T] * wp.\,(x :\oplus U.[-c,c])\,.(wp.\,(y :\oplus U.[-d,d])\,. \\
\quad ([|x-y|>T]*0 + [\neg\,(|x-y|>T)]*1)) \\
\quad + [\neg\,(|x-y|>T)]))
\end{array}
$$

$\equiv$ simple algebra

$$
\begin{array}{l}
wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,. \\
\quad ([|x-y|>T] * wp.\,(x :\oplus U.[-c,c])\,.(wp.\,(y :\oplus U.[-d,d])\,.\,([\neg\,(|x-y|>T)])) \\
\quad + [\neg\,(|x-y|>T)]))
\end{array}
$$

$\equiv$ definition of q

$$
\begin{array}{l}
wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,. \\
\quad ([|x-y|>T] * (1-q) + [\neg\,(|x-y|>T)]))
\end{array}
$$

$\equiv$ linearity, q not dependant on x or y

$$
\begin{array}{lll}
 & (1-q) * wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,.[|x-y|>T] \\
+ & wp.\,(x :\oplus U.[-a,a])\,.(wp.\,(y :\oplus U.[-b,b])\,.[\neg\,(|x-y|>T)]))
\end{array}
$$

$\equiv$ definition of p

$$(1-q) * p + (1-p)$$

$\equiv$ definition of p and q

$$
\left(\begin{array}{ll}
 & [c<d-T] * \frac{T}{d} \\
+ & [d-T\le c\le d+T] * \frac{1}{4cd} * \left(-c^2 + 2c\,(d+T) - (d-T)^2\right) \\
+ & [c>d+T] * \frac{T}{c}
\end{array}\right)
$$

$$
* \left(\begin{array}{ll}
 & [a<b-T] * \left(1-\frac{T}{b}\right) \\
+ & [b-T\le a\le b+T] * \frac{1}{4ab} * (a+b-T)^2 \\
+ & [a>b+T] * \left(1-\frac{T}{a}\right)
\end{array}\right)
$$

$$
+ \left(\begin{array}{ll}
 & [a<b-T] * \frac{T}{b} \\
+ & [b-T\le a\le b+T] * \frac{1}{4ab} * \left(-a^2 + 2a\,(b+T) - (b-T)^2\right) \\
+ & [a>b+T] * \frac{T}{a}
\end{array}\right)
$$

The probability of the pitch monitor triggering a reset is $1 - ((1-q) * p + (1-p))$, which is rearranged to get pq, i.e.

$$
\left(\begin{array}{ll}
 & [a<b-T] * \left(1-\frac{T}{b}\right) \\
+ & [b-T\le a\le b+T] * \frac{1}{4ab} * (a+b-T)^2 \\
+ & [a>b+T] * \left(1-\frac{T}{a}\right)
\end{array}\right)
$$

$$
* \left(\begin{array}{ll}
 & [c<d-T] * \left(1-\frac{T}{d}\right) \\
+ & [d-T\le c\le d+T] * \frac{1}{4cd} * (c+d-T)^2 \\
+ & [c>d+T] * \left(1-\frac{T}{c}\right)
\end{array}\right) .
$$

For example, if a and b were one and c and d were two, a tolerance T of one would result in a reset probability of

$$\left(\begin{array}{ll} & [1<0]*0 \\ + & [0\leq 1\leq 2]*\frac{1}{4} \\ + & [1>2]*0 \end{array}\right) * \left(\begin{array}{ll} & [2<1]*\frac{1}{2} \\ + & [1\leq 2\leq 3]*\frac{9}{16} \\ + & [2>3]*\frac{1}{2} \end{array}\right),$$

which reduces down to $\frac{9}{64}$.

The analysis of the pitch monitor concludes with a brief discussion about the refinement notions present in the case study. The model of the pitch monitor in Figure 8.7 could be considered to be an implementation of a more abstract specification of the pitch monitor such as

$$\mathsf{abort}\ {}_r\oplus\ \mathsf{skip}\ ,$$

where r is the probability of a reset action. The program representing the pitch monitor is an example of a data refinement of the above when the probability r is (greater than or) equal to pq as defined earlier in the section. The pitch monitor model has a lot more details about the internal workings of the program, but both of these programs exhibit the same external behaviour. They both either fail with some probability, triggering a reset action, or do nothing. Another type of refinement could be used for the individual sensor units in the situation, where a sensor can fail to return a result. This is discussed in more detail in Section 8.4.2.

8.4 Further Exploration

In this section two interesting directions for extending the analysis of the monitoring voter pattern are briefly explored. The first is in extending the reset logic to a third sensor, which then provides the opportunity for richer strategies for deciding when to reset the sensors. These strategies are discussed and one of these is analysed in detail. The second direction to be explored is that of considering the sensors as individual units that can fail and be refined by more reliable sensors. Section 8.4.2 discusses how this can be modelled in sGCL and gives example refinement proofs.

8.4.1 Adding a third sensor

If three sensors are used in the monitoring voter pattern, a number of different options for determining when a reset action should be triggered are available. With two sensors (as modelled in Section 8.2) there is only one value that can be compared to the tolerance (the difference between the two sensor readings). However, when there are three sensors, say x, y, and z, there are three differences to consider, $|x-y|$, $|x-z|$ and $|y-z|$. This leads to a number of alternative reset strategies:

$x \quad :\oplus \quad U.[0,1];$
$y \quad :\oplus \quad U.[0,1];$
$z \quad :\oplus \quad U.[0,1];$

if $|x-y| > T \vee |x-z| > T \vee |y-z| > T$ then
  abort
else
  skip
fi

– x, y, z represent the error in the reading on sensors x, y and z respectively
– T represents the maximum difference tolerated between the two sensor readings
– $|i|$ represents the absolute value of variable i

Figure 8.8: A monitoring voter with three replicated sensors

1. reset when at least one difference is greater than the tolerance,
2. reset when at least two differences are greater than the tolerance,
3. reset when all three differences are greater than the tolerance.

Note that the three differences are not independent of each other, therefore the analysis of the three sensor case is more involved than simply repeating the analysis of two sensors three times.

In this section the strategy of resetting when at least one difference is greater than the tolerance is analysed as an example. For illustration purposes the three sensors are assumed to be replicated and all follow a uniform distribution on $[0,1]$. This also allows comparison to the two sensor case studied in Section 8.2. The sGCL model for three sensors is given in Figure 8.8. The changes from the two sensor case are the extra sensor error value and the more complicated condition for abortion.

The three sensor model is also analysed in sGCL using the post-expectation of $[true]$, to give the probability that a reset action is not triggered. The formal proof follows below:

$wp.prog.[true]$
$\equiv$ definition of $prog$
$wp.(x :\oplus U.[0,1]; y :\oplus U.[0,1]; z :\oplus U.[0,1];$
  if $|x-y| > T \vee |x-z| > T \vee |y-z| > T$ then abort else skip fi$).[true]$
$\equiv$ sequential composition
$wp.\,(x :\oplus U.[0,1])\,.(wp.\,(y :\oplus U.[0,1])\,.(wp.\,(z :\oplus U.[0,1])\,.$
  $(wp.\,($if $|x-y| > T \vee |x-z| > T \vee |y-z| > T$ then abort else skip fi$)\,.[true])))$

$\equiv$ conditional

$$
\begin{array}{l}
wp.\,(x :\oplus U.[0,1])\,.(wp.\,(y :\oplus U.[0,1])\,.(wp.\,(z :\oplus U.[0,1])\,. \\
\quad ([|x-y| > T \vee |x-z| > T \vee |y-z| > T] * wp.\mathsf{abort}.[true] \\
\quad + [\neg\,(|x-y| > T \vee |x-z| > T \vee |y-z| > T)] * wp.\mathsf{skip}.[true])))
\end{array}
$$

$\equiv$ abortion, identity and definition of *true*

$$
\begin{array}{l}
wp.\,(x :\oplus U.[0,1])\,.(wp.\,(y :\oplus U.[0,1])\,.(wp.\,(z :\oplus U.[0,1])\,. \\
\quad ([|x-y| > T \vee |x-z| > T \vee |y-z| > T] * 0 \\
\quad + [\neg\,(|x-y| > T \vee |x-z| > T \vee |y-z| > T)] * 1)))
\end{array}
$$

$\equiv$ simple algebra

$$
\begin{array}{l}
wp.\,(x :\oplus U.[0,1])\,.(wp.\,(y :\oplus U.[0,1])\,.(wp.\,(z :\oplus U.[0,1])\,. \\
\quad [|x-y| \leq T \wedge |x-z| \leq T \wedge |y-z| \leq T]))
\end{array}
$$

$\equiv$ stochastic assignment

$$
wp.\,(x :\oplus U.[0,1])\,.\left(wp.\,(y :\oplus U.[0,1])\,.\left(\int_{[|x-y|\leq T \wedge |x-z|\leq T \wedge |y-z|\leq T]} \mathrm{d}U.[0,1]\right)\right)
$$

$\equiv$ integration theory

$$
\begin{array}{l}
wp.\,(x :\oplus U.[0,1])\,. \\
\left(
wp.\,(y :\oplus U.[0,1])\,.\left(
\begin{array}{lll}
 & [x < T \wedge y < x] * \displaystyle\int_0^{y+T} 1\ \mathrm{d}z & \\
+ & [x < T \wedge x \leq y \leq T] * \displaystyle\int_0^{x+T} 1\ \mathrm{d}z & \\
+ & [x < T \wedge T < y \leq x+T] * \displaystyle\int_{y-T}^{x+T} 1\ \mathrm{d}z & \\
+ & [T \leq x \leq 1-T \wedge x-T \leq y \leq x] * \displaystyle\int_{x-T}^{y+T} 1\ \mathrm{d}z & \\
+ & [T \leq x \leq 1-T \wedge x \leq y \leq x+T] * \displaystyle\int_{y-T}^{x+T} 1\ \mathrm{d}z & \\
+ & [x > 1-T \wedge y > x] * \displaystyle\int_{y-T}^{1} 1\ \mathrm{d}z & \\
+ & [x > 1-T \wedge 1-T \leq y \leq x] * \displaystyle\int_{x-T}^{1} 1\ \mathrm{d}z & \\
+ & [x > 1-T \wedge x-T \leq y < 1-T] * \displaystyle\int_{x-T}^{y+T} 1\ \mathrm{d}z &
\end{array}
\right)
\right)
\end{array}
$$

$\equiv$ integration theory and simple algebra (define α as this post-expectation)

$$wp.\,(x :\oplus U.[0,1])\,.\left(\begin{array}{ll} & [x<T\wedge y<x]*(y+T)\\ +\ & [x<T\wedge x\le y\le T]*(x+T)\\ +\ & [x<T\wedge T<y\le x+T]*(x-y+2T)\\ +\ & [T\le x\le 1-T\wedge x-T\le y\le x]*(y-x+2T)\\ +\ & [T\le x\le 1-T\wedge x\le y\le x+T]*(x-y+2T)\\ +\ & [x>1-T\wedge y>x]*(1+T-y)\\ +\ & [x>1-T\wedge 1-T\le y\le x]*(1+T-x)\\ +\ & [x>1-T\wedge x-T\le y<1-T]*(y-x+2T)\end{array}\right)$$

$\equiv$ stochastic assignment and definition of α above

$$wp.\,(x :\oplus U.[0,1])\,.\left(\int_\alpha \mathrm{d}U.[0,1]\right)$$

$\equiv$ integration theory definition of α

$$wp.\,(x :\oplus U.[0,1])\,.\left(\begin{array}{ll} & [x<T]*\left(\begin{array}{ll} & \int_0^x (y+T)\ \mathrm{d}y\\ +\ & \int_x^T (x+T)\ \mathrm{d}y\\ +\ & \int_T^{x+T} (x-y+2T)\ \mathrm{d}y\end{array}\right)\\ +\ & [T\le x\le 1-T]*\left(\begin{array}{ll} & \int_{x-T}^x (y-x+2T)\ \mathrm{d}y\\ +\ & \int_x^{x+T} (x-y+2T)\ \mathrm{d}y\end{array}\right)\\ +\ & [x>1-T]*\left(\begin{array}{ll} & \int_x^1 (1+T-y)\ \mathrm{d}y\\ +\ & \int_{1-T}^x (1+T-x)\ \mathrm{d}y\\ +\ & \int_{x-T}^{1-T} (y-x+2T)\ \mathrm{d}y\end{array}\right)\end{array}\right)$$

$\equiv$ integration theory and algebra (define β as this post-expectation)

$$wp.\,(x :\oplus U.[0,1])\,.\left(\begin{array}{ll} & [x<T]*\left(T^2+2Tx\right)\\ +\ & [T\le x\le 1-T]*\left(3T^2\right)\\ +\ & [x>1-T]*\left(T^2+2T-2Tx\right)\end{array}\right)$$

$\equiv$ stochastic assignment and definition of β above

$$\int_\beta \mathrm{d}U.[0,1]$$

$\equiv$ integration theory

$$\int_0^T \left(T^2+2Tx\right)\ \mathrm{d}x+\int_T^{1-T} 3T^2\ \mathrm{d}x+\int_{1-T}^1 \left(T^2+2T-2Tx\right)\ \mathrm{d}x$$

$\equiv$ $3T^2-2T^3$ integration theory and algebra

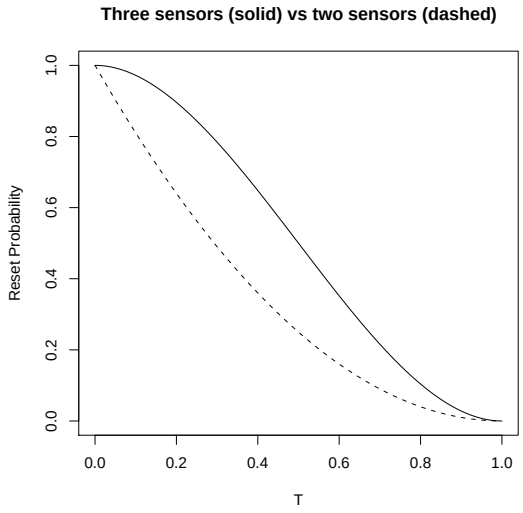

Figure 8.9: Probability of a reset action for three (solid) and two (dashed) sensors

The probability of at least one of the differences being out of tolerance is $1 - 3T^2 + 2T^3$. This is a reasonable result because the probability of a reset action decreases as the tolerance increases. This is shown in Figure 8.9, which also shows how the reset probability compares to the two sensor case. The reset strategy analysed for the three sensor case is rather strict, therefore a reset action is more likely for this scenario than for the two sensor case.

The volumes required for the analysis of the other strategies are not inherently more difficult to analyse, but require breaking the integration down into even further sections. The cross section of the volumes for the different strategies is illustrated in Figure 8.10. The numbers indicate how many differences are out of tolerance in each area of the cross-section. The equations indicate the planes that bound these areas. The key part of the volume for the case where all differences have to be within tolerance is a hexagonal prism. When a reset action is taken when at least two differences are outside the tolerance the cross section for analysis becomes a six pointed star. This adds a further six triangular prisms to the volume to be integrated (although in the case where sensors are replicated only one of these need be analysed as a symmetry argument could be used), plus the sections at either end where the limits of the probability distribution are reached. Therefore the analysis of the other strategies is left as an exercise for the interested reader.

8.4.2 Sensor refinement

This section explores the sGCL models and refinements for sensors that can fail. When a sensor fails it no longer produces errors according to the given probability distribution, it can in theory return any number. This is analogous to a software routine not

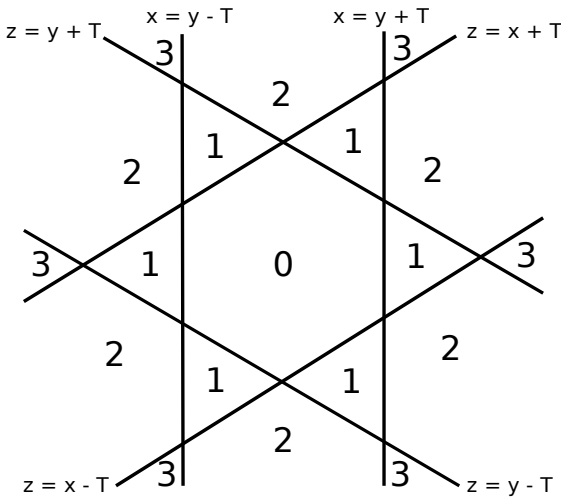

Figure 8.10: The cross-section of the volume for the different three sensor strategies

terminating, and as such the abort program is used to represent this situation in sGCL. For example, the error x given by a sensor that has an error distribution of μ whilst it is functioning correctly, but can fail with some probability $1 - p$, can be modelled in sGCL as follows

$$x :\oplus \mu \,{}_p\!\oplus \mathsf{abort}\ .$$

Such a sensor could be replaced by another sensor that has a lower chance $(1 - q)$ of failing, but the same error distribution. This would be a valid refinement (see Definition 6.2) in sGCL because the greatest pre-expectation of any expectation, $wp.prog.\beta$, must be greater in the replacement sensor. This is shown formally below for $p < q$

$$\begin{array}{lll}
 & prog \sqsubseteq prog' & \\
\text{iff} & wp.prog.\beta \leq wp.prog'.\beta & \text{Definition 6.2} \\
\text{iff} & wp.\left(x :\oplus \mu \,{}_p\!\oplus \mathsf{abort}\right).\beta \leq wp.\left(x :\oplus \mu \,{}_q\!\oplus \mathsf{abort}\right).\beta & \text{definition of } prog,\ prog' \\
\text{iff} & & \text{probabilistic choice} \\
 & \quad p * wp.\left(x :\oplus \mu\right).\beta + (1-p) * wp.\mathsf{abort}.\beta & \\
 & \leq \quad q * wp.\left(x :\oplus \mu\right).\beta + (1-q) * wp.\mathsf{abort}.\beta & \\
\text{iff} & p * wp.\left(x :\oplus \mu\right).\beta \leq q * wp.\left(x :\oplus \mu\right).\beta & \text{abortion} \\
\text{iff} & true & \text{algebra, } p < q
\end{array}$$

More complicated refinements are also possible, for example the refining sensor $prog'$ may have a different error distribution to $prog$. In this case it is only a valid refinement when the probability of any outcome in $prog'$ is at least that of the probability of the same outcome in $prog$. For example, assume that $prog$ has the uniform error distribution over $[0, 1]$, but can fail with probability 0.1. This is equivalent to

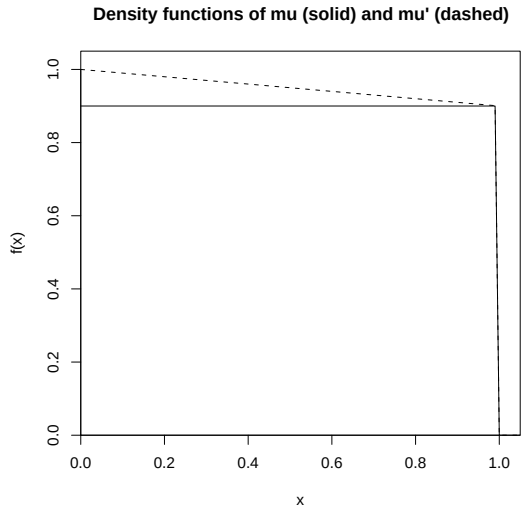

Figure 8.11: The probability density functions of mu (solid) and mu' (dashed)

$$\mu.[a,b] := \begin{cases} \frac{9}{10} * (b-a) & \text{if } 0 \leq a < b \leq 1 \ , \\ 0 & \text{otherwise .} \end{cases}$$

An example $prog'$ that refines $prog$ has an error distribution as follows

$$\mu'.[a,b] := \begin{cases} (b-a) - \frac{1}{20}\left(b^2 - a^2\right) & \text{if } 0 \leq a < b \leq 1 \ , \\ 0 & \text{otherwise .} \end{cases}$$

The pdfs of these two error distributions are shown in Figure 8.11.

To show that $prog \sqsubseteq prog'$ the equivalent (Lemma 6.12), relational definition of refinement may be used (Definition 6.7). The non-trivial case (when $0 \leq a < b \leq 1$) is shown below:

	$\mu \sqsubseteq \mu'$	
iff	$\mu.[a,b] \leq \mu'.[a,b]$	Definition 6.7, for arbitrary $0 \leq a < b \leq 1$
iff	$\frac{9}{10} * (b-a) \leq (b-a) - \frac{1}{20}\left(b^2 - a^2\right)$	definition of μ, μ'
iff	$\left(b^2 - a^2\right) \leq 2\,(b-a)$	algebra
iff	$(b+a)\,(b-a) \leq 2\,(b-a)$	algebra
iff	$true$	algebra, $0 \leq a < b \leq 1$

Therefore the program $prog$ that assigns variable x according to μ is refined by the program $prog'$ that assigns x according to μ'.

Refinement of sensors allows a specification of a minimum acceptable error distribution and failure rate to be given. The implementation of the system can then use any sensor that can be shown to be a refinement of the specification. Any properties shown

to hold for the specification will be preserved in the implementation.

8.5 Evaluation

In this chapter a new design pattern, the monitoring voter, has been described and its novel aspects have been analysed. The analysis focussed on the reset logic and how this affected the probability of requiring a reset action for a variety of sensors. This pattern was then applied to a real case study, that of an aircraft pitch monitor. In this situation resetting the sensors is a significant inconvenience as it requires the aeroplane to be in level flight before it can happen. Therefore it is important to know, and thus be able to minimise, the probability of a reset action being required. The opportunities for refinement and further analysis of the monitoring voter were also explored.

The main aim of the case study was to illustrate the use of sGCL, particularly in the analysis of properties involving multiple continuous probability distributions. In a discrete approximation of such analysis, a trade-off would be required between the accuracy of the approximation and the size of the state space (and thus the calculations required). However, in sGCL the integration techniques handle this complexity and the analysis is relatively straightforward without losing any accuracy. Therefore such analysis is a strength of sGCL. The case study of the monitoring voter required the analysis of the difference of two variables. It is not possible to analyse this difference without considering the two variables simultaneously. The fact that the two variables were assigned according to continuous probability distributions meant that this provided a good example to illustrate the strength of sGCL in this analysis. The pitch monitor example provided a real life scenario to demonstrate the importance of this analysis.

A secondary aim of the case study was to provide and demonstrate the use of a methodological contribution, namely that of design patterns. The use of design patterns made the analysis of the pitch monitor easier to understand. The monitoring voter could be analysed independently, and the results of this analysis were fed into the analysis of the pitch monitor. This made the analysis of the pitch monitor relatively easy as the analysis from the monitoring voters was directly transferable. This reuse of analysis demonstrates the advantage of using design patterns, thus achieving the secondary aim of the case study.

A further aim of the case study was to demonstrate the use of refinement in sGCL. This was briefly discussed for the pitch monitor and explored in more detail for sensors in the further exploration. However, it would be interesting to investigate more complex refinement relations as further work.

There were some aspects of sGCL that the case study did not provide the scope for examining. In particular, loops were not considered at all. This was not held to be a priority because the techniques demonstrated still apply to finite loops, just with more complex conditional and sequential composition combinations. However, it would

be interesting to explore loop invariant reasoning (as was illustrated in Chapter 3 for discrete probability) in sGCL as further work.

Overall the case study provided some useful insight, and provided a good idea of the use and benefits of sGCL. However, the study has also indicated areas that could be explored in more depth in future work.

Chapter 9

Conclusions

This chapter summarises the work presented in this thesis (Section 9.1) and discusses further challenges in the area of stochastic model-based specification languages. The thesis aims (Section 1.2.1) are revisited and the extent to which they have been met is discussed (Section 9.2). The opportunities for future research are then explored (Section 9.3).

9.1 Summary

The need for probability in formal methods, as a means of providing rigorous analysis of the dependability aspects of computing systems, was discussed in Chapter 1. In particular, the need for continuous probability was highlighted as it is often difficult to express and analyse fault behaviour without it. For example, faults may occur according to an exponential distribution over time, or the reading error of an analogue sensor may follow a normal distribution. The rigour associated with the specification and analysis of systems using model-based specification languages is highly desirable for providing the assurance required for the development of dependable systems. However, such languages have little or no support for continuous probability. This thesis developed a new *stochastic* model-based specification language as a first step towards addressing that gap. This new language allows variables to be assigned according to continuous probability distributions. Restrictions have been defined to ensure the consistency of the language, in particular the state space is restricted to bounded intervals of reals, recursion is finite, and non-determinism is excluded.

In Part 1 of the thesis the role of probability in formal methods was explored in more detail. The key concepts needed to understand the thesis were presented (Chapter 2). This included a discussion of related research, which concluded that whilst one or two of the languages examined provide reasonable support for continuous probability, they lack some desirable features. In particular, the support for refinement (a formal development approach from abstract specifications to concrete implementations) or non-determinism

(an important abstraction technique) has not been explored. Further, the practical application of these languages (through the use of examples or case studies) has not been demonstrated.

Part 1 continued with a demonstration of the practical use of probability in model-based specification languages, through a case study on flash memory (Chapter 3). The aim of the case study was to illustrate how probabilistic systems can be described and analysed in a formal way. The case study used the language pGCL, as this forms the basis of the new language developed in the thesis. Using pGCL meant that only discrete probability could be used, which led into a discussion of the limitations of the formalism and the kinds of questions that are more easily answered using continuous probability.

Part 1 concluded with a further case study based on an emergency brake (Chapter 4) to illustrate the features of two contrasting formalisms (PRISM and pB), and to prototype a stochastic model-based specification language. The prototype language that was presented (stochastic Event-B) aims to combine the benefits of model-based specification languages and continuous probability. The complexities involved in defining a full formal definition of stochastic Event-B were discussed, in particular the fact that combining probability and non-determinism is not a trivial problem. This allowed the identification of a suitable simpler language (pGCL) for a formal stochastic extension in Part 2 of this thesis. The key advantage of pGCL is that non-determinism is modelled explicitly, and can therefore be removed without having to re-define the whole language.

Part 2 provided the formal development of sGCL, a model-based specification language that supports continuous probability. Before the syntax and semantics of the language were defined, some more technical background material (Chapter 5) was required. This included: an introduction to measure theory, which provides the basis of the relational semantics of sGCL; an overview of the Giry monad for the sequential composition of stochastic programs; the definition of the Kantorovich distance between two measures, for compactness arguments; and a more detailed look at pGCL, in particular the relational semantics. These form the main building blocks of the relational semantics of sGCL.

The main chapter of Part 2 (and the thesis) described a deterministic sGCL (Chapter 6). This chapter defined the syntax, transformer and relational semantics of sGCL, which constitutes the main contribution of this thesis. The transformer semantics is similar to that presented by Kozen [54], but includes an explicit definition for stochastic assignment and more intuitive syntax. Important healthiness conditions, which ensure that all programs have meanings, were shown to hold in sGCL. The relational semantics of sGCL was based on measure theory and related to the transformer semantics through the Riesz representation theorem. A refinement ordering for sGCL programs was also defined, for both the transformer and relational semantics, and its practical applications were discussed. The chapter concluded with a comparison of sGCL to the related work presented in Part 1.

Part 2 continued with an exploration of non-determinism and continuous probability. The sGCL language definition was extended to include demonic non-determinism (Chapter 7). This included the definition of the syntax, transformer and relational semantics of a non-deterministic sGCL. However, the mathematics required to demonstrate the consistency of the transformer and relational semantics is challenging and not fully developed. Approaches to the consistency proofs were discussed and the remaining challenges for their successful implementation were described. As this discussion is very technical, the majority of the material was presented in Appendix D, with just an overview offered in the main body of this thesis. Adding non-determinism enabled a further notion of refinement in sGCL, which was also explored. Once more, the chapter concluded with a discussion of how the state of the art was furthered by the research presented. The author believes that this chapter presented the first attempt to combine demonic non-determinism and arbitrary continuous probability distributions in a formal specification language.

In Part 3 the practical use of sGCL was demonstrated through the use of a final case study on an aeroplane pitch monitor (Chapter 8). The sGCL modelling approach was integrated with the established engineering practice of design patterns. A new design pattern relating to dependability was defined, and the probabilistic elements of it were analysed in sGCL. The new design pattern was then applied to the case study, in which it was used to simplify the analysis of the pitch monitor. Opportunities for refinement in the case study were discussed, including the definition of an abstract dependability specification, and the refinement of faulty sensors with (suitable) more reliable alternatives. The chapter concluded with an evaluation of the suitability of the case study for exploring the capabilities of sGCL, and of the performance of sGCL in analysing the case study. The evaluation identified opportunities for further research, particularly in more detailed explorations of refinement and iteration in sGCL.

9.2 Evaluation

Section 1.2.1 contained a discussion of aims that the research presented in this thesis should satisfy. These included some general aims regarding the exploration of continuous probability in model-based specification languages and some more specific aims that the language developed should satisfy. In this section, these aims are revisited and the extent to which they have been satisfied is evaluated. The applicability of sGCL for modelling and analysing various classes of faults is also briefly discussed.

9.2.1 General aims

The overall vision of this thesis was to equip developers with the ability to reason about systems with (continuous) probabilistic elements, in the context of model-based formal methods for developing dependable systems. Recall (Section 1.2.1) that to achieve this

vision the following general aims were identified:

- to explore the role of probability in model-based specification languages, in particular continuous probability;
- to develop a model-based specification language that supports reasoning about continuous probability;
- to demonstrate reasoning about dependability properties using the new language;
- to examine how the new language fits into software engineering practice, such as the use of design patterns.

These are examined in turn and the extent to which they have been met is evaluated.

Exploring the role of probability in model-based specification languages

In Part 1 of this thesis a wide range of probabilistic and stochastic formal methods were examined (Section 2.3). These ranged from probabilistic process algebras to stochastic petri nets and probabilistic model checkers. As the main focus of this research is on model-based specification languages, these were examined in more detail. A number of probabilistic model-based specification languages were identified and discussed, with pGCL and pB explored in more detail through the use of case studies (Chapters 3 and 4). The story was different for stochastic model-based specification languages, however. The state of the art for these was found to be rather limited, with only a few attempts to combine continuous probability and model-based specification languages. Even then, the research on continuous probability tended to be an aside to the main argument (on discrete probability). The literature was found to be lacking on any practical applications of continuous probability in model-based specification languages. Issues such as refinement and non-determinism had also not been explored in detail. This led to a wider consideration of continuous probability in formal methods, including the use of the emergency brake case study (Chapter 4) to explore the support for continuous probability in the PRISM model checker. However, even in the wider context of formal methods, the support for continuous probability was found to be limited to specific distributions, typically the exponential distribution. The concluding section of Chapter 3 discussed the need to support a wider range of probability distributions in modelling languages.

In conclusion, the role of probability in model-based specification languages, and more generally in formal methods, has been explored in depth through the available literature and use of case studies. However, as the state of the art is continually progressing, there will inevitably be further contributions to consider.

Developing a stochastic model-based specification language

A stochastic model-based specification language, sGCL, was developed in Part 2 of this thesis. Specifically, a deterministic version of sGCL was developed in full as described in Chapter 6, and the extension of this to include non-determinism was examined in detail in Chapter 7. However, the completion of the non-deterministic version is beyond the scope of this thesis. In particular, the mathematical foundations needed to prove the consistency of the transformer and relational semantics are not readily available for the non-deterministic version. The extent to which the specific requirements of the language are met is discussed in more detail below (Section 9.2.2). Here, some more general observations about the language developed are discussed.

Whilst the deterministic version of sGCL is a fully defined model-based specification language that supports continuous probability, a number of restrictions were required in order to achieve this. The most crucial of these was the restriction of the state space. In order to prove the consistency between the transformer and relational semantics, the state space had to be restricted to closed intervals of the real numbers, $[a, b]$ for finite $a, b \in \mathbb{R}$. This means that standard distributions such as the exponential or normal distribution cannot be modelled in sGCL. Truncated versions of them may be used instead, but the truncated versions do not have the useful properties of the original distributions, such as the memoryless property of the exponential distribution used in CTMCs. However, in spite of this restriction, some interesting opportunities for analysis exist, as was demonstrated in the case study on sGCL (Chapter 8). A second restriction of sGCL was the limitation to finite loops. This was required to prove the healthiness conditions of sGCL, but means that steady state properties of CTMCs cannot be analysed. However, without the ability to model the exponential distribution, this further restriction has minimal impact.

In conclusion, a model-based specification language has been developed that supports continuous probability and key features such as abstraction, refinement and formal analysis of probabilistic properties. However, the state space and use of recursion had to be restricted in order to achieve this.

Demonstrating reasoning about dependability properties in sGCL

The case study (Chapter 8) was used to demonstrate the application of sGCL, in particular, for reasoning about dependability properties. The case study had two parts, the first analysed a monitoring voter pattern for improving dependability, whilst the second applied this pattern for the analysis of an aeroplane pitch monitor. The first part demonstrated how continuous probability could be modelled and analysed in sGCL. In particular, it demonstrated how complex interactions between random variables can be handled relatively straightforwardly using the integration techniques inherent in sGCL. The practical application of such analysis to the pitch monitor demonstrated the need

for the capabilities of sGCL. The purpose of refinement in sGCL was also discussed with respect to the pitch monitor and some example refinement proofs were given.

There is certainly more scope for demonstrating how to reason about dependability properties in sGCL, in particular for features such as looping and more detailed refinement chains. However, the case study presented in Chapter 8 provided a good indication of the benefits of the reasoning capabilities provided by sGCL, and how such analysis can be performed.

Examining how sGCL fits into software engineering practice

The case study in Chapter 8 examined how the use of sGCL would fit in with aspects of established software engineering practice. In particular, the probabilistic aspects of a design pattern aiming at improving dependability were analysed in sGCL. Design patterns are important in software engineering to capture elements of design that frequently occur. The case study successfully illustrated how the use and analysis of a dependability design pattern in sGCL simplified the reasoning about an aeroplane pitch monitor. However, there are many more methods and tools used in software engineering, so there is plenty of scope for further investigation in this area. Also, for sGCL to be incorporated into recommended practice, tool support would be essential, and is currently unavailable.

9.2.2 Language aims

As the language that was developed (sGCL) is a key feature of this thesis, a deeper evaluation of its capabilities with respect to the original aims (Section 1.2.1) is given here. The aims stated that sGCL should:

- include explicit support for continuous probability distributions, so that variables may be assigned values according to a continuous probability distribution;
- allow formal analysis and proof of (probabilistic) properties over systems containing continuous probability distributions, for example dependability attributes such as reliability and safety;
- provide means for abstraction and refinement of computer-based systems containing continuous probability distributions;
- be underpinned by a consistent mathematical foundation.

A further desirable aim suggested that sGCL should:

- support non-determinism, where the program can branch according to some external decisions beyond the control of the program.

The extent to which these aims have been met is discussed below. Unless otherwise stated, the language being evaluated below is the deterministic version of sGCL.

Explicit support for continuous probability distributions

The language sGCL introduced a new program construct $:\oplus$ that allows a variable to be assigned a value according to a continuous probability distribution. This is described in detail in Chapter 6. The continuous probability distribution is modelled as a measure, but it is straightforward (Section 6.2) to translate the notation of cdfs and standard probability distributions to that of measures. The meaning of this construct and others in sGCL programs was defined through the transformer and relational semantics. However, the range of continuous probability distributions that could be modelled in sGCL had to be restricted according to the state space limitations described above (Section 9.2.1). Only distributions defined over closed intervals of the real numbers can be modelled in sGCL. Nonetheless, there is explicit support for continuous probability distributions in sGCL.

Support for formal analysis of probabilistic properties

The transformer semantics (and to some extent the relational semantics) of sGCL enables the formal analysis of probabilistic properties. This analysis is typically carried out using the weakest pre-condition rules given in Section 6.3, although demonstrating valid refinements of very abstract specifications provides an alternative approach. The practical application of the weakest pre-condition rules for analysing dependability properties was demonstrated in the case study in Chapter 8.

Support for abstraction and refinement

There are a number of ways that sGCL supports abstraction. These include: abstract constructs such as skip and abort; the use of sub-probability measures; and the inclusion of probabilistic choice. It would be undesirable to have a construct in a programming language that does nothing (skip), or worse, causes the program to not terminate or fail (abort). However, such abstract constructs are useful in sGCL to model the situations where something good or something bad may occur. Similarly, the distribution of a random variable may not be fully known, but it may still be desirable to analyse it to some extent. In this case, the use of sub-probability (where the total probability defined is less than one) is useful, but it is unlikely to be found in any implementation. Finally, discrete probability can be used as an abstraction for continuous probability, by partitioning the state space into regions of particular interest. This abstraction is possible in sGCL because it supports discrete probability as well as continuous probability.

There are two notions of refinement in sGCL (Section 6.5). The first is a refinement ordering for continuous probability measures. This essentially increases the definedness of a measure by increasing the total probability defined (without reducing the probability of any outcome in the abstract measure). The second is the notion of data refinement, where abstract constructs are replaced by more concrete ones (as long as the more

concrete construct can replace the more abstract one without detection in terms of functional properties) [61]. The practical application of these notions of refinement (and the use of abstraction) is illustrated in the case study on sGCL (Chapter 8). A further abstraction mechanism and refinement notion is also explored in sGCL (Section 7.4). This requires the use of non-determinism, which is discussed in detail below.

Overall, sGCL provides good support for abstraction and refinement of computer-based systems containing continuous probability distributions.

A consistent mathematical foundation

The meaning of sGCL is defined in terms of the transformer semantics. The Riesz representation theorem is used to provide an equivalent and consistent relational semantics in terms of measure theory (Section 6.4). To ensure that the Riesz representation theorem applies to all the programs that can be defined in sGCL, a number of healthiness conditions were shown to hold in the transformer semantics (Section 6.3.1). To prove these, the state space had to be restricted to compact intervals of real numbers and recursion had to be limited to finite loops. However, with these restrictions in place, sGCL has a consistent mathematical foundation.

Support for non-determinism

The extension of the deterministic version of sGCL to include non-determinism was examined in Chapter 7. The syntax, transformer and relational semantics for a non-deterministic sGCL has been defined. However, the proofs needed to demonstrate a consistent mathematical foundation are significantly challenging when non-determinism is included in sGCL. In particular, the mathematical theorems and lemmas required for such proofs are not readily available. Whilst, some effort has been spent exploring alternative proof approaches, the definition of a (proven) consistent non-deterministic sGCL is beyond the scope of this thesis.

Therefore the support for non-determinism in sGCL has been explored, but further research is required to fully define a non-deterministic version of sGCL.

9.2.3 Discussion of the fault classes sGCL may apply to

The case study (Chapter 8) demonstrated how sGCL could be used to model and analyse faulty sensors. This section briefly discusses what other types of faults may be analysed in sGCL, with reference to the main fault classes identified by Avizienis et al. [9]. A thorough examination of the fault classes applicable to sGCL is left as further work.

Avizienis et al. [9] identify three major (and partially overlapping) groupings of faults. These are:

- **Development faults** – faults that occur during system development, system maintenance or generation of procedures for operating or maintaining the system;

- **Physical faults** – faults that affect hardware;
- **Interaction faults** – faults that are external, i.e. originate outside the system boundary and propagate errors into the system by interaction or interference.

Of these categories, physical faults is the most obvious candidate for modelling in sGCL. This includes the sensor faults from the case study, floating point arithmetic rounding errors as discussed by Hasan et al. [36] and failure rates of hardware, all of which may follow continuous probability distributions (although some failure rate distributions require an unbounded state space, so would need the state space restriction to be lifted first). In addition, physical faults that follow discrete distributions (such as the number of messages lost in a network) may be approximated (using limit theories) by continuous distributions for large systems (but such limiting distributions tend to include infinity, so would probably need the state space restriction to be lifted first).

It is harder to see how sGCL could be applied to development faults. Whilst it is possible that a continuous distribution could model the rate at which development faults are introduced to a system (based on factors such as the work environment, procedures followed etc.), getting sufficient data to accurately model this would not be easy.

Interaction faults are more interesting, in particularly those involving interactions with humans. Quantification of accidental human faults (such as operator errors) is a challenging area as the factors involved in such faults are usually complex. Therefore it is probably premature to try to model or analyse these in sGCL as it may be hard to identify suitable abstractions in terms of continuous probability distributions. However, malicious human faults may follow a clearer pattern and be easier to quantify, for example the time between service requests in a denial of service attack might be usefully modelled and analysed (in sGCL) using continuous probability. It would be interesting to explore this example in future work.

9.3 Further Work

Based on the above evaluation of the work presented in this thesis, some suggestions for further research (in the area of continuous probability in model-based specification languages) are discussed. These extensions fall into three categories: those relating to the language itself; those relating to applying the language and integrating it with other methods; and those relating to tool support. These are discussed in more detail below.

9.3.1 Language extensions

A number of limitations of sGCL were discussed in the evaluation. In particular: the restriction of the state space to closed intervals of real numbers; the restriction of recursion to finite loops; the exclusion of demonic non-determinism. Lifting these restrictions

would turn sGCL into an even more powerful language. However, each of these is a challenging area for further research as discussed below.

Broadening the state space

The restriction of the state space to closed intervals of real numbers is required through the use of the Riesz representation theorem to relate the transformer and relational semantics. Therefore, to lift this restriction, alternative approaches for relating the transformer and relational semantics would need to be explored. Perhaps a good starting point for this would be to examine the semantics for a subset of measures. Considering exponential distributions, in particular, may provide some useful insight as this has been explored already in the context of process algebras [37] and model checkers [57]. Another possibility may be to explicitly define the relational to transformer embedding and transformer to relational retraction as for the non-deterministic version of sGCL (Section 7.3). This would require more proof effort to determine the consistency of the semantics, but perhaps the embedding and retraction could be defined so that the state space restriction is no longer required.

Infinite loops

The restriction of recursion to finite loops was required to show the healthiness conditions of sGCL, in particular continuity. This is because it is not possible to show, in general, that the limit of a sequence of continuous functions is itself continuous. However, there may be a subset of infinite loops that continuity may be shown to hold for. For example, McIver et al. [61] allow loops in pGCL that "almost certainly" terminate. This means that the loop terminates with probability one, but it is impossible to put a finite bound on the number of iterations that will be executed before termination. It would be interesting to investigate how such loops could fit into the definition of sGCL.

Non-determinism

In Chapter 7, the extension of sGCL to include non-determinism was examined. However, to fully define a non-deterministic sGCL, further research is required. In particular, alternative proof strategies for the consistency proofs need to be investigated. An alternative strategy for one of the proofs was outlined in Appendix D, which appears to be a promising direction for future work. The other proof is trickier, however. Possible directions for exploration include a different approach to converting measures to Euclidean space, or examining alternative metric spaces for the compactness arguments.

Once non-determinism has been resolved, it would be possible to explore the original idea, of extending Event-B with continuous probability (Section 4.4), in more detail. It is, at best, difficult and inelegant to define a deterministic subset of Event-B. Therefore, it seems pointless to revisit the development of Stochastic Event-B, before a solution

has been found for handling continuous probability and non-determinism in a consistent manner.

An especially challenging language extension would lift the state space restriction *and* include non-determinism. The promising compactness arguments in the consistency proofs rely on the fact that the underlying state space is also compact, i.e. a closed interval of the reals. Until these language extensions have been explored separately, it is hard to predict how a combination of the two may be achieved.

9.3.2 Methodological extensions

In the evaluation above, two key areas for further exploration with respect to the application of sGCL were identified. These were: integrating sGCL with alternate software engineering methods; and more detailed case studies to illustrate and explore the use of sGCL. These extensions may not appear to be as technically challenging as the language extensions, but are vital to encourage the wider use of sGCL. These areas are discussed in turn below.

Integration with software engineering and dependability mechanisms

In the case study (Chapter 8) the use of sGCL to analyse design patterns was explored. The two approaches worked well together to simplify the analysis of an aeroplane pitch monitor. It would be interesting to investigate whether there are other design patterns that could also benefit from such integration with sGCL. These may include: dependability patterns, such as voting or recovery blocks; patterns that add more realism to a model, such as noise on sensor readings; or more general software engineering patterns.

It would also be beneficial to examine whether there are other software engineering or dependability practices that could benefit from the analysis provided by sGCL. For example, could sGCL (or some similar language) assist in the quantitative analysis of fault trees (in situations where there are complex interactions between faults occurring according to continuous probability distributions)? Another area where a language like sGCL may be useful is in the analysis of risk associated with hazards, e.g. in the safety methodology HazOp. These risks can be hard to quantify, and possibly the refinement approach of sGCL could assist with such quantification. It is envisaged that this would work by starting with an abstract model of the world, and iteratively adding complexities, until a realistic valuation of the risk is determined.

There are many opportunities to integrate the benefits provided by a language, such as sGCL, with those of existing methods. The above only aims to provide a flavour of the possibilities.

Further case studies

Further case studies would aid in the understanding of the capabilities of sGCL, and how best to use it according to its strengths. Particular features of sGCL that would benefit from further exploration are loops and refinement chains. Therefore it would be beneficial to determine suitable case studies to examine these features in more detail.

The flash case study (Chapter 3) provides the opportunity for exploring loops. However, the continuous probability distributions that were identified to be inherent in the flash memory problem require the inclusion of infinity in the state space, i.e. cannot be modelled by closed intervals of real numbers. Therefore the state space restrictions need to be lifted to explore this properly (although truncated versions of these distributions could be used as an approximation).

Another option would be to extend the case study on the aeroplane pitch monitor (Chapter 8). More details could be added to the scenario through a series of refinement steps. For example, the timing elements of the case study could be added, as these were abstracted away from in the analysis presented in this thesis.

Finally, case studies could be used to further explore the fault classes that are suitable for modelling in sGCL (see Section 9.2.3). For example, it could be interesting to explore how suitable sGCL is for modelling and analysing the malicious behaviour in a denial of service attack.

9.3.3 Tool support

No modelling language has a chance of industrial uptake without tool support. Therefore it is an important avenue of further work to develop tools that support modelling, and particularly reasoning, in sGCL.

Proving properties of sGCL models requires significant expertise. Therefore this is a key area in which tool support should be provided. Luckily, there already exist some building blocks for this in the HOL theorem prover. A mechanism of pGCL has already been completed in HOL [41] and there is also support for reasoning about expectation properties of (continuous) measures [36]. Therefore, it is a realistic goal to mechanise sGCL in HOL.

Other aspects of developing sGCL models would benefit from tool support. Validation of sGCL models (through syntax and type checking, and the use of interpreters to exercise sGCL statements) is an obvious role of tool support. To interpret (or simulate) a model that contains (continuous) probability requires a means of obtaining random observations from the probability distributions involved. This functionality could be provided by statistical software (such as R[1]). It is worth noting that a probabilistic model needs to be exercised a large number of times (with the results of these runs presented graphically, for example) to obtain an accurate picture of its behaviour.

[1] See http://www.r-project.org/.

More ambitious features may assist users in developing sGCL models, such as a library of patterns (like the monitoring voter from Chapter 8, or those discussed in Section 9.3.2) that guide the user towards designing more dependable systems, or integration with other tools such as model checkers.

9.4 Closing statements

The support for continuous probability distributions in model-based specification languages has been explored to new depths in this thesis. Whilst significant challenges have been encountered, a new language has been developed that supports rigorous modelling and analysis of computer-based systems containing continuous probability. The capabilities and usefulness of this language have been demonstrated through the use of a case study. The limitations of the language have also been evaluated and from this many interesting directions for further research have been identified.

Appendix A

Proving the Flash Memory Loop Invariant

This appendix provides further details of the proof work required to analyse the flash filestore system described in Chapter 3. In particular, this requires showing that the expression given in Formula 3.1 is in fact an invariant of the loop described in Figure 3.2.

Recall (Formula 3.1) that the claimed invariant of the loop is as follows:

$$m + n +$$

$$\sum_{e=N-m}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e + \sum_{e=N-n}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e$$

This will be referred to as Inv in the following for convenience. To show that this is in fact an invariant of the loop in Section 2.1 it is required to show that:

$$[G] * Inv \leq wp.body.Inv \ . \tag{A.1}$$

The proof proceeds as follows:

$$\begin{array}{lll} & wp.body.Inv & \\ \equiv & wp.\left(\left(m := m+1 \ {}_{\frac{1}{2}}\!\oplus n := n+1\right);\ e := e+1\right).Inv & \text{definition of } body \\ \equiv & & \text{sequential composition} \\ & wp.\left(m := m+1 \ {}_{\frac{1}{2}}\!\oplus n := n+1\right).\left(wp.\left(e := e+1\right).Inv\right) & \end{array}$$

$\equiv$ simple substitution and definition of Inv

$$wp.\ \left(m := m+1 \ {}_{\frac{1}{2}}\!\oplus\! \ n := n+1\right).$$
$$\left(\begin{array}{l} m+n+\displaystyle\sum_{e+1=N-m}^{2N-(m+n+1)} (e+1)\left(\begin{array}{c} e+1-1 \\ N-m-1 \end{array}\right)\left(\frac{1}{2}\right)^{e+1} \\ +\displaystyle\sum_{e+1=N-n}^{2N-(m+n+1)} (e+1)\left(\begin{array}{c} e+1-1 \\ N-n-1 \end{array}\right)\left(\frac{1}{2}\right)^{e+1} \end{array}\right)$$

$\equiv$ arithmetic (change of variables $e' = e+1$)

$$wp.\ \left(m := m+1 \ {}_{\frac{1}{2}}\!\oplus\! \ n := n+1\right).$$
$$\left(\begin{array}{l} m+n+\displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e\left(\begin{array}{c} e-1 \\ N-m-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \\ +\displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e\left(\begin{array}{c} e-1 \\ N-n-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \end{array}\right)$$

$\equiv$ probabilistic choice substitution

$$\tfrac{1}{2} * wp.\,(m := m+1).\left(\begin{array}{l} m+n+\displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e\left(\begin{array}{c} e-1 \\ N-m-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \\ +\displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e\left(\begin{array}{c} e-1 \\ N-n-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \end{array}\right)$$

$$+\quad \tfrac{1}{2} * wp.\,(n := n+1).\left(\begin{array}{l} m+n+\displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e\left(\begin{array}{c} e-1 \\ N-m-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \\ +\displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e\left(\begin{array}{c} e-1 \\ N-n-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \end{array}\right)$$

$\equiv$ simple substitution

$$\tfrac{1}{2} * \left(\begin{array}{l} m+1+n+\displaystyle\sum_{e=N-(m+1)}^{2N-((m+1)+n+1)} e\left(\begin{array}{c} e-1 \\ N-(m+1)-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \\ +\displaystyle\sum_{e=N-n}^{2N-((m+1)+n+1)} e\left(\begin{array}{c} e-1 \\ N-n-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \end{array}\right)$$

$$+\quad \tfrac{1}{2} * \left(\begin{array}{l} m+n+1+\displaystyle\sum_{e=N-m}^{2N-(m+(n+1)+1)} e\left(\begin{array}{c} e-1 \\ N-m-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \\ +\displaystyle\sum_{e=N-(n+1)}^{2N-(m+(n+1)+1)} e\left(\begin{array}{c} e-1 \\ N-(n+1)-1 \end{array}\right)\left(\frac{1}{2}\right)^{e} \end{array}\right)$$

$\equiv$ algebra

$$m+n+1+ \quad \tfrac{1}{2} * \left(\begin{array}{l} \displaystyle\sum_{e=(N-m)-1}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ (N-m-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-n}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-m}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=(N-n)-1}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ (N-n-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \right)$$

The first and third summations of the big expression above are considered first:

$$\tfrac{1}{2} * \left(\begin{array}{l} \displaystyle\sum_{e=(N-m)-1}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ (N-m-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-m}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \right)$$

$\equiv$ algebra (shifting the sums by 1)

$$\begin{array}{l} \tfrac{1}{2} * \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-2 \\ (N-m-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^{e-1} \\ + \quad \tfrac{1}{2} * \displaystyle\sum_{e=(N-m)+1}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-2 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^{e-1} \end{array}$$

$\equiv$ algebra (bringing the $\frac{1}{2}$ into the sums)

$$\begin{array}{l} \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-2 \\ (N-m-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^{e} \\ + \displaystyle\sum_{e=(N-m)+1}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-2 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^{e} \end{array}$$

$\equiv$ algebra (taking the first term out of the first sum)

$$\begin{array}{l} \displaystyle\sum_{e=(N-m)+1}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-2 \\ (N-m-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^{e} \\ + \displaystyle\sum_{e=(N-m)+1}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-2 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^{e} + (N-m-1) * \frac{1}{2}^{N-m} \end{array}$$

$\equiv$ algebra (combining sums and combinations, using the rule ${}_aC_b = {}_{a-1}C_b + {}_{a-1}C_{b-1}$)

$$\displaystyle\sum_{e=(N-m)+1}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^{e} + (N-m-1) * \frac{1}{2}^{N-m}$$

$\equiv$ algebra (putting the floating term back into the sum)

$$\sum_{e=N-m}^{2N-(m+n+1)} (e-1) * \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e$$

$\equiv$ algebra (splitting into sums with e and -1)

$$\sum_{e=N-m}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e - \sum_{e=N-m}^{2N-(m+n+1)} \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e$$

A similar argument follows for the second and fourth sums of the previous expression. Using the those results the following can now be shown:

$$m+n+1+ \quad \tfrac{1}{2} * \left(\begin{array}{l} \displaystyle\sum_{e=(N-m)-1}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ (N-m-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-n}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-m}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=(N-n)-1}^{2N-(m+n+1)-1} e \left(\begin{array}{c} e-1 \\ (N-n-1)-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \right)$$

$\equiv$ algebra (see above)

$$\begin{array}{ll} m+n+1+ & \left(\begin{array}{l} \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \right) \\ - & \left(\begin{array}{l} \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-n}^{2N-(m+n+1)} \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array} \right) \end{array}$$

$\equiv$ algebra (the second half of this is the sum of the pdf † of the distribution = 1)

$$\begin{array}{ll} m+n+ & \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\ + & \displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e \left(\begin{array}{c} e-1 \\ N-n-1 \end{array} \right) \left(\frac{1}{2}\right)^e \end{array}$$

$\equiv$ Inv by definition

The pdf remark † can be informally reasoned about as follows. These two summations expand to give the probabilities of all the possible ways in which the remaining number of iterations of the loop can occur after $m+n$ iterations have already happened

(based on the negative binomial distribution)[1]. As the loop is guaranteed to terminate in one of these ways (when either n or m reaches N) these probabilities must sum to one. Note that the guard of the loop ensures that $m < N \wedge n < N$.

When reasoning about loops it is also important to show that the termination values are as expected, i.e. that:

$$Inv(m, N, e) = m + N \ , \tag{A.2}$$

and

$$Inv(N, n, e) = N + n \ . \tag{A.3}$$

This is proved formally for the former (a similar argument follows by symmetry for the latter) as shown below:

$$\begin{array}{ll}
& Inv(m, N, e) \\
\equiv & \text{definition of } Inv \\
& m + N + \displaystyle\sum_{e=N-m}^{2N-(m+N+1)} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\
& \quad + \displaystyle\sum_{e=N-N}^{2N-(m+N+1)} e \left(\begin{array}{c} e-1 \\ N-N-1 \end{array} \right) \left(\frac{1}{2}\right)^e \\
\equiv & \text{arithmetic} \\
& m + N + \displaystyle\sum_{e=N-m}^{N-m-1} e \left(\begin{array}{c} e-1 \\ N-m-1 \end{array} \right) \left(\frac{1}{2}\right)^e + \sum_{e=0}^{N-m-1} e \left(\begin{array}{c} e-1 \\ -1 \end{array} \right) \left(\frac{1}{2}\right)^e \\
\equiv & \text{arithmetic}^\dagger \\
& m + N + 0 + 0 \\
\equiv & \text{arithmetic} \\
& m + N
\end{array}$$

Note that for the line marked †, the combination ${}_nC_{-1}$ is not really zero, but undefined. To overcome this problem, the loop invariant Inv could be modified to include the guard $[m < N \wedge n < N]$. It would be trivial to include this in all of the calculations, but it has been omitted to provide a cleaner presentation.

Finally, the expected lifetime e of the system can be derived by evaluating the loop invariant Inv at initialisation, i.e. $Inv(0, 0, e)$. This analysis proceeds as follows:

$$\begin{array}{ll}
& Inv(0, 0, e) \\
\equiv & \text{definition of } Inv \\
& 0 + 0 + \displaystyle\sum_{e=N-0}^{2N-(0+0+1)} e \left(\begin{array}{c} e-1 \\ N-0-1 \end{array} \right) \left(\frac{1}{2}\right)^e + \sum_{e=N-0}^{2N-(0+0+1)} e \left(\begin{array}{c} e-1 \\ N-0-1 \end{array} \right) \left(\frac{1}{2}\right)^e
\end{array}$$

[1]Note that this expression is exactly that of the second half of the expectation, without the multiplication by e to get the expected value of e.

$\equiv$ arithmetic

$$\sum_{e=N}^{2N-1} e \left(\begin{array}{c} e-1 \\ N-1 \end{array} \right) \left(\frac{1}{2}\right)^e + \sum_{e=N}^{2N-1} e \left(\begin{array}{c} e-1 \\ N-1 \end{array} \right) \left(\frac{1}{2}\right)^e$$

$\equiv$ arithmetic

$$2 * \sum_{e=N}^{2N-1} e \left(\begin{array}{c} e-1 \\ N-1 \end{array} \right) \left(\frac{1}{2}\right)^e$$

This concludes the formal analysis of the wear-levelling algorithm for flash filestores presented in Chapter 3.

Appendix B

Emergency Brake Models and Proofs

This appendix provides the full models and proofs for the emergency brake case study discussed in Chapter 4. The PRISM models are presented first, these are followed by the pB models and proofs, and finally the stochastic Event-B models and proofs are presented.

B.1 PRISM Models

This section provides the PRISM models of the emergency brake case study as discussed in Section 4.2. The first model contains all the functionality in one module, whereas the second model is separated into modules for requesting and applying the emergency brake. Both models have the same reward structure, which is used in Section 4.2 to analyse the likelihood of ending up in the unsafe failure state.

B.1.1 Version 1

// First PRISM model of the emergency brake (EB) scenario, all functionality in one module
// Author: Zoe Andrews
// Version: 1.0 (18 July 2008)

ctmc

// Parameters:

// The brake is requested this many times per hour
const double request_rate = 0.1;

```
// The brake fails in a safe way (EB applied not commanded) this many times per hour
const double safe_fail_rate = 0.001;

// The probability that the brake fails in an unsafe way (EB commanded, not applied)
const double unsafe_fail_prob = 0.01;

module EB

// State variables
commanded : bool init false; // Whether the EB has been requested or not
applied : bool init false; // Whether the EB is applied or not

// A successful request for the EB
[request_success] commanded=false & applied=false ->
    request_rate*(1-unsafe_fail_prob) : commanded'=true & applied'=true;

// An unsuccessful request for the EB (ie an unsafe failure)
[unsafe_failure] commanded=false & applied=false ->
    request_rate*unsafe_fail_prob : commanded'=true;

// A safe failure occurs, ie the emergency brake is applied without a command
[safe_failure] commanded=false & applied=false -> safe_fail_rate : applied'=true;

endmodule

// Reward structures
rewards "unsafe failures"
commanded=true & applied=false : 1;
endrewards
```

B.1.2 Version 2

```
// Second PRISM model of the emergency brake (EB) scenario
// Functionality separated into request and application modules
// Author: Zoe Andrews
// Version: 1.0 (4th December 2008)

ctmc

// Parameters:
```

```
// The brake is requested this many times per hour
const double request_rate = 0.1;

// The brake fails in a safe way (EB applied not commanded) this many times per hour
const double safe_fail_rate = 0.001;

// The probability that the brake fails in an unsafe way (EB commanded, not applied)
const double unsafe_fail_prob = 0.01;

module EB_request

// State variables
commanded : bool init false; // Whether the EB has been requested or not

// Synchronises with the request action in the EB_application module
// Models the act of requesting the EB
[request] commanded=false & applied=false -> request_rate : commanded'=true;

endmodule


module EB_application

// State variables
applied : bool init false; // Whether the EB is applied or not

// A safe failure occurs, ie the emergency brake is applied without a command
[safe_failure] commanded=false & applied=false -> safe_fail_rate : applied'=true;

// Synchronises with the request action in the EB_request module
// Once a request has occurred the EB is applied with probability
// 1 - the unsafe failure probability (and remains false with prob unsafe_fail_prob)
[request] true -> 1 - unsafe_fail_prob : applied'=true + unsafe_fail_prob : applied'=false;

endmodule

// Reward structures
rewards "unsafe failures"
commanded=true & applied=false : 1;
endrewards
```

Note that the analysis of the PRISM models was described in Section 4.2 and is not repeated here.

B.2 pB Models

This section provides the pB models of the emergency brake case study as discussed in Section 4.3. The first model ignores the timing aspects of the case study and just considers the relative likelihood of each the events occurring, whereas the second model includes a *time* variable to approximate the advancing of time. The analysis of these models is given in Appendix B.3.

B.2.1 Option 1

// Version 1 based on Embedded Markov Chain approach
// (What happens next? Timing aspects of the case study are ignored)

MACHINE $EmergencyBrakeV1()$

CONSTANTS
p_us, p_safe // Probabilities of unsafe and safe failure events respectively

PROPERTIES
$p\_us \in REAL \wedge p\_safe \in REAL$

VARIABLES
// c and n record historical data for analysis
// n represents total runs of the model
// c represents number of runs resulting in unsafe failures
$EB\_command, EB\_applied, c, n$

INVARIANT
$EB\_command \in BOOL \wedge EB\_applied \in BOOL \wedge c \in NAT \wedge n \in NAT$

EXPECTATIONS
// Expected proportion of unsafe failures is always at most p_max
$real(0) \leq real(n) \times p\_max - real(c)$

INITIALISATION
// Initially the brake is neither requested nor applied
$EB\_command := FALSE \;||\; EB\_applied := FALSE \;||\; c := 0 \;||\; n := 0$

OPERATIONS

$EB\_Request \mathrel{\widehat{=}}$ // The EB is requested
PRE
$EB\_command = FALSE \wedge EB\_applied = FALSE$
THEN
$(EB\_command, c := TRUE, c+1)\ {}_{p\_us}\oplus$
$(EB\_command, EB\_applied := TRUE, TRUE)$
END;

$main \mathrel{\widehat{=}}$ // Main operation: Either an EB request or safe failure occurs
PRE
$EB\_command = FALSE \wedge EB\_applied = FALSE$
THEN
$(EB\_applied := TRUE\ {}_{p\_safe}\oplus\ EB\_Request())$
$||\ n := n+1$
END

END

B.2.2 Option 2

// Version 2 based on discrete clock approach
// (What happens in the next time unit? Discrete approximation to timing aspects)

MACHINE $EmergencyBrakeV2()$

CONSTANTS
// Probabilities of unsafe failure, safe failure and request events respectively
p_us, p_safe, p_req

PROPERTIES
$p\_us \in REAL \wedge p\_safe \in REAL \wedge p\_req \in REAL$

VARIABLES
// c and time record historical data for analysis
// time represents the number of time units passed
// c represents number of runs resulting in unsafe failures
$EB\_command, EB\_applied, c, time$

INVARIANT
$EB\_command \in BOOL \wedge EB\_applied \in BOOL \wedge c \in NAT \wedge time \in NAT$

EXPECTATIONS
// Expected rate of unsafe failures is always at most p_max
$real(0) \leq real(time) \times p\_max - real(c)$

INITIALISATION
$EB\_command := FALSE \;||\; EB\_applied := FALSE \;||\; c := 0 \;||\; time := 0$

OPERATIONS

$EB\_Request \mathrel{\widehat{=}}$ // The EB is requested
PRE
$EB\_command = FALSE \wedge EB\_applied = FALSE$
THEN
$(EB\_command, c := TRUE, c+1) \;\; {}_{p\_us}\oplus$
$(EB\_command, EB\_applied := TRUE, TRUE)$
END;

$main \mathrel{\widehat{=}}$ // Main operation: Either an EB request, a safe failure or nothing occurs
PRE
$EB\_command = FALSE \wedge EB\_applied = FALSE$
THEN
$(EB\_applied := TRUE \;\; {}_{p\_safe}\oplus \;\; (EB\_Request() \;\; {}_{p\_req}\oplus \;\; SKIP))$
$||\; time := time + 1$
END

END

B.3 pB Expectation Analysis

This section provides the analysis of the safety property of the emergency brake case study in the pB models given above and discussed in Section 4.3. Recall (Section 4.1) that the safety property states that "$unsafe\ situation \leq \lambda_{max}/hour$". This property was recorded in the expectations clause of the pB models above. This section gives the formal proof that the pB models respect these expectations. The analysis is given in turn for the two different pB models, starting with the model that ignores the timing aspects of the case study, followed by the model that uses a time variable to approximate the advancing of time.

B.3.1 Option 1

Recall (Section 4.3) that the expectation of the first pB model is as follows:

$$Inv \mathrel{\widehat{=}} n \times p\_max - c \;.$$

First it is required to show that the initialisation establishes the expectation. This

involves proving the following:

$$0 \leq [Initialisation]\, Inv\ .$$

Consider the right hand side of the inequality:

$$\begin{array}{ll}
& [Initialisation]\, Inv \\
\equiv & \qquad\qquad \text{definition of } Initialisation \\
& [EB_command, EB_applied, c, n := FALSE, FALSE, 0, 0]\, Inv \\
\equiv & \qquad\qquad \text{definition of } Inv \\
& [EB_command, EB_applied, c, n := FALSE, FALSE, 0, 0]\, (n \times p_max - c) \\
\equiv & \quad 0 \times p_max - 0 \qquad\qquad \text{simple substitutions} \\
\equiv & \quad 0 \qquad\qquad \text{arithmetic}
\end{array}$$

Thus the initialisation procedure establishes the initial lower bound of the expectation.

It is also required to show that the operations respect the expectation. Only the *main* operation is examined for this because this is the only operation that is considered to be external[1]. To show that the *main* operation respects the expectation, the following needs to be proved:

$$Inv \leq [main]\, Inv\ .$$

Consider the right hand side of this inequality:

$$\begin{array}{ll}
& [main]\, Inv \\
\equiv & \qquad\qquad \text{definition of } main \\
& \left[\begin{array}{l} (EB_applied := TRUE \ {}_{p_safe}\!\oplus\ EB_Request\,()) \\ || \\ n := n + 1 \end{array}\right] Inv \\
\equiv & \qquad\qquad \text{definition of } EB_Request \text{ and } Inv \\
& \left[\begin{array}{l} \left(\begin{array}{l} EB_applied := TRUE \\ {}_{p_safe}\!\oplus \\ \left(\begin{array}{l} EB_command, c := TRUE, c + 1 \\ {}_{p_us}\!\oplus \\ EB_command, EB_applied := TRUE, TRUE \end{array}\right) \end{array}\right) \\ || \\ n := n + 1 \end{array}\right] (n \times p_max - c)
\end{array}$$

[1]The other operations are considered to be internal (they are called by *main*) and are only modelled as separate operations for a clearer presentation.

$\equiv$ parallel substitution with ${}_{p\_safe}\oplus$

$$\left[\begin{array}{l} \left(\begin{array}{l} EB\_applied := TRUE \\ || \\ n := n+1 \end{array}\right) \\ {}_{p\_safe}\oplus \\ \left(\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ {}_{p\_us}\oplus \\ EB\_command, EB\_applied := TRUE, TRUE \end{array}\right) \\ || \\ n := n+1 \end{array}\right) \end{array}\right] (n \times p\_max - c)$$

$\equiv$ probabilistic choice substitution ${}_{p\_safe}\oplus$

$$p\_safe \times \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ n := n+1 \end{array}\right] (n \times p\_max - c)$$

$$+(1 - p\_safe) \times \left[\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ {}_{p\_us}\oplus \\ EB\_command, EB\_applied := TRUE, TRUE \end{array}\right) \\ || \\ n := n+1 \end{array}\right] (n \times p\_max - c)$$

$\equiv$ parallel substitution with ${}_{p\_us}\oplus$

$$p\_safe \times \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ n := n+1 \end{array}\right] (n \times p\_max - c)$$

$$+(1 - p\_safe) \times \left[\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ || \\ n := n+1 \end{array}\right) \\ {}_{p\_us}\oplus \\ \left(\begin{array}{l} EB\_command, EB\_applied := TRUE, TRUE \\ || \\ n := n+1 \end{array}\right) \end{array}\right] (n \times p\_max - c)$$

$\equiv$ probabilistic choice substitution ${}_{p\_us}\oplus$

$$
\begin{array}{l}
p\_safe \times \\
\quad \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ n := n+1 \end{array}\right] (n \times p\_max - c) \\
\\
+ (1 - p\_safe) \times \\
\quad \left(\begin{array}{l}
p\_us \times \\
\quad \left[\begin{array}{l} EB\_command, c := TRUE, c+1 \\ || \\ n := n+1 \end{array}\right] (n \times p\_max - c) \\
\\
+ (1 - p\_us) \times \\
\quad \left[\begin{array}{l} EB\_command, EB\_applied := TRUE, TRUE \\ || \\ n := n+1 \end{array}\right] (n \times p\_max - c)
\end{array}\right)
\end{array}
$$

$\equiv$ parallel substitution and simple substitution

$$
\begin{array}{lll}
 & p\_safe \times ((n+1) \times p\_max - c) & \\
+ & (1 - p\_safe) \times \left(\begin{array}{ll} & p\_us \times ((n+1) \times p\_max - (c+1)) \\ + & (1 - p\_us) \times ((n+1) \times p\_max - c) \end{array}\right)
\end{array}
$$

To maintain the expectation it is required that $Inv \leq [main]\, Inv$, thus (from the above substitutions) that:

$$
\begin{array}{lll}
(n \times p\_max - c) & \leq & p\_safe \times ((n+1) \times p\_max - c) \\
 & & + \quad (1 - p\_safe) \times \left(\begin{array}{ll} & p\_us \times ((n+1) \times p\_max - (c+1)) \\ + & (1 - p\_us) \times ((n+1) \times p\_max - c) \end{array}\right)
\end{array}
$$

$\equiv$ arithmetic

$$
(1 - p\_safe) \times p\_us \quad \leq \quad p\_max
$$

This concludes the analysis of the expectation for the first pB model, the inequality found above was discussed in Section 4.3.

B.3.2 Option 2

Recall (Section 4.3) that the expectation of the second pB model is as follows:

$$Inv \mathrel{\widehat{=}} time \times p\_max - c \ .$$

First it is required to show that the initialisation establishes the expectation. This involves proving the following:

$$0 \leq [Initialisation]\, Inv \ .$$

Consider the right hand side of the inequality:

$$
\begin{array}{ll}
 & [\mathit{Initialisation}]\,\mathit{Inv} \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } \mathit{Initialisation} \\
 & [\mathit{EB\_command}, \mathit{EB\_applied}, c, \mathit{time} := \mathit{FALSE}, \mathit{FALSE}, 0, 0]\,\mathit{Inv} \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } \mathit{Inv} \\
 & [\mathit{EB\_command}, \mathit{EB\_applied}, c, \mathit{time} := \mathit{FALSE}, \mathit{FALSE}, 0, 0]\,(\mathit{time} \times \mathit{p\_max} - c) \\
\equiv & \quad 0 \times \mathit{p\_max} - 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{simple substitutions} \\
\equiv & \quad 0 \qquad \text{arithmetic}
\end{array}
$$

Thus the initialisation procedure establishes the initial lower bound of the expectation.

It is also required to show that the operations respect the expectation. As with the first pB model, only the *main* operation is examined for this because this is the only operation that is considered to be external. To show that the *main* operation respects the expectation, the following needs to be proved:

$$\mathit{Inv} \leq [\mathit{main}]\,\mathit{Inv}\ .$$

Consider the right hand side of this inequality:

$$
\begin{array}{ll}
 & [\mathit{main}]\,\mathit{Inv} \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } \mathit{main} \\
 & \left[\begin{array}{ll} & (\mathit{EB\_applied} := \mathit{TRUE}\ {}_{p\_safe}\!\oplus (\mathit{EB\_Request}\,()\ {}_{p\_req}\!\oplus \mathsf{skip})) \\ || & \\ & \mathit{time} := \mathit{time} + 1 \end{array}\right] \mathit{Inv} \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } \mathit{EB\_Request} \\
 & \left[\begin{array}{ll} & \left(\begin{array}{l} \mathit{EB\_applied} := \mathit{TRUE} \\ {}_{p\_safe}\!\oplus \\ \left(\begin{array}{l} \left(\begin{array}{l} \mathit{EB\_command}, c := \mathit{TRUE}, c+1 \\ {}_{p\_us}\!\oplus \\ \mathit{EB\_command}, \mathit{EB\_applied} := \mathit{TRUE}, \mathit{TRUE} \end{array}\right) \\ {}_{p\_req}\!\oplus \\ \mathsf{skip} \end{array}\right) \end{array}\right) \\ || & \\ & \mathit{time} := \mathit{time} + 1 \end{array}\right] \mathit{Inv}
\end{array}
$$

$\equiv$ parallel substitution with $_{p\_safe}\oplus$

$$\left[\begin{array}{l} \left(\begin{array}{l} EB\_applied := TRUE \\ || \\ time := time + 1 \end{array}\right) \\ {}_{p\_safe}\oplus \\ \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ {}_{p\_us}\oplus \\ EB\_command, EB\_applied := TRUE, TRUE \end{array}\right) \\ {}_{p\_req}\oplus \\ \quad \mathsf{skip} \end{array}\right) \\ || \\ time := time + 1 \end{array}\right) \end{array}\right] Inv$$

$\equiv$ probabilistic choice substitution $_{p\_safe}\oplus$

$$p\_safe \times \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ time := time + 1 \end{array}\right] Inv$$

$$+(1 - p\_safe) \times \left[\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ {}_{p\_us}\oplus \\ EB\_command, EB\_applied := TRUE, TRUE \end{array}\right) \\ {}_{p\_req}\oplus \\ \quad \mathsf{skip} \end{array}\right) \\ || \\ time := time + 1 \end{array}\right] Inv$$

$\equiv$ parallel substitution with $_{p\_req}\oplus$

$$p\_safe \times \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ time := time + 1 \end{array}\right] Inv$$

$$+(1 - p\_safe) \times \left[\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ {}_{p\_us}\oplus \\ EB\_command, EB\_applied := TRUE, TRUE \end{array}\right) \\ || \\ time := time + 1 \end{array}\right) \\ {}_{p\_req}\oplus \\ \left(\begin{array}{l} \mathsf{skip} \\ || \\ time := time + 1 \end{array}\right) \end{array}\right] Inv$$

$\equiv$ probabilistic choice substitution $_{p\_req}\oplus$

$$
\begin{array}{l}
p\_safe \times \\
\quad \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ time := time + 1 \end{array}\right] Inv \\
\\
+ (1 - p\_safe) \times \\
\quad \left(\begin{array}{l} p\_req \times \\ \quad \left[\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c + 1 \\ {}_{p\_us}\oplus \\ EB\_command, EB\_applied := TRUE, TRUE \end{array}\right) \\ || \\ time := time + 1 \end{array}\right] Inv \\ \\ + (1 - p\_req) \times \\ \quad \left[\begin{array}{l} \mathsf{skip} \\ || \\ time := time + 1 \end{array}\right] Inv \end{array}\right)
\end{array}
$$

$\equiv$ parallel substitution with $_{p\_us}\oplus$

$$
\begin{array}{l}
p\_safe \times \\
\quad \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ time := time + 1 \end{array}\right] Inv \\
\\
+ (1 - p\_safe) \times \\
\quad \left(\begin{array}{l} p\_req \times \\ \quad \left[\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c + 1 \\ || \\ time := time + 1 \end{array}\right) \\ {}_{p\_us}\oplus \\ \left(\begin{array}{l} EB\_command, EB\_applied := TRUE, TRUE \\ || \\ time := time + 1 \end{array}\right) \end{array}\right] Inv \\ \\ + (1 - p\_req) \times \\ \quad \left[\begin{array}{l} \mathsf{skip} \\ || \\ time := time + 1 \end{array}\right] Inv \end{array}\right)
\end{array}
$$

$\equiv$ probabilistic choice substitution $_{p\_us}\oplus$

$$
\begin{array}{l}
p\_safe \times \\
\quad \left[\begin{array}{l} EB\_applied := TRUE \\ || \\ time := time + 1 \end{array}\right] Inv \\
\\
+ (1 - p\_safe) \times \\
\quad \left(\begin{array}{l} p\_req \times \\ \quad \left(\begin{array}{l} p\_us \times \\ \quad \left[\begin{array}{l} EB\_command, c := TRUE, c+1 \\ || \\ time := time + 1 \end{array}\right] Inv \\ \\ + (1 - p\_us) \times \\ \quad \left[\begin{array}{l} EB\_command, EB\_applied := TRUE, TRUE \\ || \\ time := time + 1 \end{array}\right] Inv \end{array}\right) \\ \\ + (1 - p\_req) \times \\ \quad \left[\begin{array}{l} \mathsf{skip} \\ || \\ time := time + 1 \end{array}\right] Inv \end{array}\right)
\end{array}
$$

$\equiv$ parallel substitution, simple substitution and definition of Inv

$$
\begin{array}{l}
p\_safe \times \\
\quad ((time + 1) \times p\_max - c) \\
+ (1 - p\_safe) \times \\
\quad \left(\begin{array}{l} p\_req \times \\ \quad \left(\begin{array}{rcl} p\_us & \times & ((time + 1) \times p\_max - (c + 1)) \\ + \quad (1 - p\_us) & \times & ((time + 1) \times p\_max - c) \end{array}\right) \\ + (1 - p\_req) \times \\ \quad ((time + 1) \times p\_max - c) \end{array}\right)
\end{array}
$$

To maintain the expectation it is required that $Inv \leq [main]\, Inv$, thus (from the above substitutions) that:

$$
\begin{array}{rcl}
(time \times p\_max - c) & \leq & p\_safe \times ((time + 1) \times p\_max - c) \\
& & + (1 - p\_safe) \times \\
& & \quad \left(\begin{array}{l} p\_req \times \\ \quad \left(\begin{array}{l} p\_us \times \\ \quad ((time + 1) \times p\_max - (c + 1)) \\ + (1 - p\_us) \times \\ \quad ((time + 1) \times p\_max - c) \end{array}\right) \\ + (1 - p\_req) \times \\ \quad ((time + 1) \times p\_max - c) \end{array}\right)
\end{array}
$$

$\equiv$ arithmetic

$$(1 - p\_safe) \times p\_req \times p\_us \quad \leq \quad p\_max$$

This concludes the analysis of the expectation for the second pB model, the inequality found above was discussed in Section 4.3.

B.4 Event-B Models

This section provides the Event-B models of the emergency brake case study as discussed in Section 4.4. First a model of the emergency brake is given in standard Event-B (with no stochastic behaviour). Then two stochastic Event-B models are presented. The first of the stochastic Event-B models uses the rate parameter to model the stochastic behaviour, whilst the second has an explicit *time* variable that is updated according to the exponential distribution. The analysis of the standard Event-B model is discussed in Section 4.4, whilst the full analysis of the stochastic Event-B models is given in Appendix B.5.

B.4.1 Standard Event-B

The standard Event-B model of the emergency brake (EB) models just the logical behaviour of the system, without any indication of its dependability attributes.

MACHINE EB

VARIABLES

`EB_applied` % Whether the EB is applied or not

`EB_command` % Whether the EB has been requested or not

INVARIANTS

`inv1` : $EB\_applied \in BOOL$

`inv2` : $EB\_command \in BOOL$

`inv3` : $EB\_command = TRUE \implies EB\_applied = TRUE$
% Safety property: if the EB is requested, then it is applied.

EVENTS

Initialisation

begin

`act1` : $EB\_applied := FALSE$

`act2` : $EB\_command := FALSE$

end

Event *Unsafe_Failure* $\widehat{=}$ % EB is requested, but not applied properly

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

act1 : $EB\_command := TRUE$

end

Event *Safe_Failure* $\widehat{=}$ % EB is applied, but has not been requested

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

act1 : $EB\_applied := TRUE$

end

Event *EB_Normal* $\widehat{=}$ % EB is requested and applied properly

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

act1 : $EB\_applied := TRUE$

act2 : $EB\_command := TRUE$

end

END

B.4.2 Stochastic Event-B option 1

The first stochastic Event-B model uses a rate parameter to update the time passed before an event occurs implicitly (see Section 4.4). Fresh variables (c and n) track the history for the analysis of the probabilistic choice statement. Note that the invariant labelled $exp1$ is really an expectation, likewise the actions labelled *rate* update time according to the exponential distribution (Event-B does not have a way of recording these new constructs).

MACHINE pEBv1

VARIABLES

EB_applied % Whether the EB is applied or not

EB_command % Whether the EB has been requested or not

c % Records number of unsafe failures

n % Records number of EB requests

INVARIANTS

inv1 : $EB\_applied \in BOOL$

inv2 : $EB\_command \in BOOL$

inv3 : $c \in \mathbb{N}$

inv4 : $n \in \mathbb{N}$

exp1 : $0 \leq \ n \times \lambda_{max} - c \times \lambda_{req}$

% Expectation recording the safety property: rate of unsafe failures is at most λ_{max}.

EVENTS

Initialisation

begin

act1 : $EB\_applied := FALSE$

act2 : $EB\_command := FALSE$

act3 : $c := 0$

act4 : $n := 0$

end

Event $Safe\_Failure \mathrel{\widehat{=}}$ % EB is applied, but has not been requested

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

rate : λ_{safe}

act1 : $EB\_applied := TRUE$

end

Event $EB\_Request \mathrel{\widehat{=}}$ % EB is requested and is either applied successfully or fails

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

rate : λ_{req}

act1 : $(EB\_command, c, n := TRUE, c + 1, n + 1) \ {}_p\oplus$
$(EB\_applied, EB\_command, n := TRUE, TRUE, n + 1)$

end

END

B.4.3 Stochastic Event-B option 2

The second stochastic Event-B model has an explicit *time* variable that is updated according to the exponential distribution to model the timing of an event (see Section 4.4). Fresh variable c tracks the history for the analysis of the probabilistic choice statement. Note that the invariant labelled $exp1$ is really an expectation (Event-B does not have a way of recording expectations).

MACHINE pEBv2

VARIABLES

`EB_applied` % Whether the EB is applied or not

`EB_command` % Whether the EB has been requested or not

`c` % Records number of unsafe failures

`time` % Records the amount of time that has passed

INVARIANTS

`inv1` : $EB\_applied \in BOOL$

`inv2` : $EB\_command \in BOOL$

`inv3` : $c \in \mathbb{N}$

`inv4` : $time \in REAL$

`exp1` : $0 \leq time \times \lambda_{max} - c$

% Expectation recording the safety property: rate of unsafe failures is at most λ_{max}.

EVENTS

Initialisation

begin

`act1` : $EB\_applied := FALSE$

`act2` : $EB\_command := FALSE$

`act3` : $c := 0$

`act4` : $time := 0.0$

end

Event $Safe\_Failure \mathrel{\widehat{=}}$ % EB is applied, but has not been requested

when

`grd1` : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

`rate` : λ_{safe}

`act1` : $EB\_applied := TRUE$

`act2` : $time := time + \mathbf{exp}(\lambda_{req} + \lambda_{safe})$

end

Event $EB\_Request \mathrel{\widehat{=}}$ % EB is requested and is either applied successfully or fails

when

grd1 : $EB\_applied = FALSE \wedge EB\_command = FALSE$

then

rate : λ_{req}

act1 : $(EB\_command, c := TRUE, c + 1)\ {}_p\oplus$
$(EB\_applied, EB\_command := TRUE, TRUE)$

act2 : $time := time + \mathbf{exp}(\lambda_{req} + \lambda_{safe})$

end

END

B.5 Stochastic Event-B Expectation Analysis

This section provides the analysis of the safety property of the emergency brake case study in the stochastic Event-B models given above and discussed in Section 4.4. Recall (Section 4.1) that the safety property states that "$unsafe\ situation \leq \lambda_{max}/hour$". This property was recorded in a special invariant labelled $exp1$ in the stochastic Event-B models above. This section gives the formal proof that the stochastic Event-B models respect these expectations. The analysis is given in turn for the two different stochastic Event-B models, starting with the model that implicitly models time through the rate parameter, followed by the model that uses an explicit assignment to update $time$.

B.5.1 Option 1

Recall (Section 4.4) that the expectation of the first stochastic Event-B model is as follows:

$$Inv \mathrel{\widehat{=}} n \times \lambda_{max} - c \times \lambda_{req} \ .$$

First it is required to show that the initialisation establishes the expectation. This involves proving the following:

$$0 \leq [Initialisation]\, Inv \ .$$

Consider the right hand side of the inequality:

$[Initialisation]\, Inv$

$\equiv$ definition of $Initialisation$

$[EB\_command, EB\_applied, c, n := FALSE, FALSE, 0, 0]\, Inv$

$\equiv$ definition of Inv

$[EB\_command, EB\_applied, c, n := FALSE, FALSE, 0, 0]\, (n \times \lambda_{max} - c \times \lambda_{req})$

$$\begin{array}{lll} \equiv & 0 \times \lambda_{max} - 0 \times \lambda_{req} & \text{simple substitutions} \\ \equiv & 0 & \text{arithmetic} \end{array}$$

Thus the initialisation event establishes the initial lower bound of the expectation.

It is also required to show that the events respect the expectation. Only the *EB_Request* event is examined here because this is the most interesting event (the $Safe\_Failure$ event trivially respects the expectation). To show that the *EB_Request* event respects the expectation, the following needs to be proved:

$$Inv \leq [EB\_Request]\, Inv \ .$$

Consider the right hand side of this inequality:

$$\begin{array}{ll} & [EB\_Request]\, Inv \\ \equiv & \text{definition of } EB\_Request \text{ and } Inv \\ & \left[\begin{array}{l} EB\_command, c, n := TRUE, c+1, n+1 \\ {}_p\oplus \\ EB\_command, EB\_applied, n := TRUE, TRUE, n+1 \end{array}\right] (n \times \lambda_{max} - c \times \lambda_{req}) \\ \equiv & \text{probabilistic choice substitution } {}_p\oplus \\ & p \times \\ & \quad [EB\_command, c, n := TRUE, c+1, n+1]\,(n \times \lambda_{max} - c \times \lambda_{req}) \\ & + (1-p) \times \\ & \quad [EB\_command, EB\_applied, n := TRUE, TRUE, n+1]\,(n \times \lambda_{max} - c \times \lambda_{req}) \\ \equiv & \text{parallel substitution and simple substitution} \\ & \qquad p \quad \times \quad ((n+1) \times \lambda_{max} - (c+1) \times \lambda_{req}) \\ & + \quad (1-p) \quad \times \quad ((n+1) \times \lambda_{max} - c \times \lambda_{req}) \end{array}$$

To maintain the expectation it is required that $Inv \leq [main]\, Inv$, thus (from the above substitutions) it is required that:

$$\begin{array}{lll} (n \times \lambda_{max} - c \times \lambda_{req}) & \leq & p \times ((n+1) \times \lambda_{max} - (c+1) \times \lambda_{req}) \\ & & + \quad (1-p) \times ((n+1) \times \lambda_{max} - c \times \lambda_{req}) \\ \equiv & & \text{arithmetic} \\ p \times \lambda_{req} & \leq & \lambda_{max} \end{array}$$

This concludes the analysis of the expectation for the first stochastic Event-B model, the inequality found above was discussed in Section 4.4.

B.5.2 Option 2

Two expectations are analysed for option 2 as the first gives an interesting result as discussed in Section 4.4.3.

Recall (Section 4.4) that the first expectation of interest (Formula 4.9) of the second stochastic Event-B model is as follows:

$$Inv \mathrel{\widehat{=}} time \times \lambda_{max} - c \ .$$

First it is required to show that the initialisation establishes the expectation. This involves proving the following:

$$0 \leq [Initialisation]\, Inv \ .$$

Consider the right hand side of the inequality:

$$\begin{array}{ll}
& [Initialisation]\, Inv \\
\equiv & \qquad\qquad \text{definition of } Initialisation \\
& [EB_command, EB_applied, c, time := FALSE, FALSE, 0, 0]\, Inv \\
\equiv & \qquad\qquad \text{definition of } Inv \\
& [EB_command, EB_applied, c, time := FALSE, FALSE, 0, 0]\, (time \times \lambda_{max} - c) \\
\equiv & \quad 0 \times \lambda_{max} - 0 \qquad\qquad \text{simple substitutions} \\
\equiv & \quad 0 \qquad\qquad \text{arithmetic}
\end{array}$$

Thus the initialisation event establishes the initial lower bound of the expectation.

It is also required to show that the events respect the expectation. Only the $EB\_Request$ event is examined here because this is the most interesting event (the $Safe\_Failure$ event trivially respects the expectation). To show that the $EB\_Request$ event respects the expectation, the following needs to be proved:

$$Inv \leq [EB\_Request]\, Inv \ .$$

Consider the right hand side of this inequality:

$$\begin{array}{ll}
& [EB_Request]\, Inv \\
\equiv & \qquad\qquad \text{definition of } EB_Request \text{ and } Inv \\
& \left[\begin{array}{l} \left(\begin{array}{l} EB_command, c := TRUE, c+1 \\ {}_p\oplus \\ EB_command, EB_applied := TRUE, TRUE \end{array} \right) \\ \quad || \\ \quad time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array} \right] (time \times \lambda_{max} - c) \\
\equiv & \qquad\qquad \text{parallel substitution with } {}_p\oplus \\
& \left[\begin{array}{l} \left(\begin{array}{l} EB_command, c := TRUE, c+1 \\ || \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array} \right) \\ {}_p\oplus \\ \left(\begin{array}{l} EB_command, EB_applied := TRUE, TRUE \\ || \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array} \right) \end{array} \right] (time \times \lambda_{max} - c)
\end{array}$$

$\equiv$ probabilistic choice substitution ${}_p\oplus$

$$p \times \left[\begin{array}{l} EB\_command, c := TRUE, c+1 \\ || \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array}\right] (time \times \lambda_{max} - c)$$

$$+ (1-p) \times \left[\begin{array}{l} EB\_command, EB\_applied := TRUE, TRUE \\ || \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array}\right] (time \times \lambda_{max} - c)$$

$\equiv$ exponential, parallel and simple substitution

$$\begin{array}{rcl} p & \times & \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - (c+1)\right) \\ + \quad (1-p) & \times & \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - c\right) \end{array}$$

To maintain the expectation it is required that $Inv \leq [main]\, Inv$, thus (from the above substitutions) it is required that:

$$\begin{array}{rcl} (time \times \lambda_{max} - c) & \leq & p \times \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - (c+1)\right) \\ & + & (1-p) \times \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - c\right) \end{array}$$

$\equiv$ arithmetic

$$p \times (\lambda_{req} + \lambda_{safe}) \leq \lambda_{max}$$

This concludes the analysis of the first expectation (Formula 4.9) for the second stochastic Event-B model. However, this provides a conservative estimate of the relationship between the parameters (see Section 4.4.3), therefore a second expectation is also analysed.

Recall (Section 4.4) that the second expectation of interest (Formula 4.11) for the second stochastic Event-B model is as follows:

$$Inv \mathrel{\widehat{=}} time \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req} + \lambda_{safe}} \times c \ .$$

First it is required to show that the initialisation establishes the expectation. This involves proving the following:

$$0 \leq [Initialisation]\, Inv \ .$$

Consider the right hand side of the inequality:

$$[Initialisation]\, Inv$$

$\equiv$ definition of $Initialisation$

$$[EB\_command, EB\_applied, c, time := FALSE, FALSE, 0, 0]\, Inv$$

$$
\begin{array}{ll}
\equiv & \text{definition of } Inv \\
& \left[\begin{array}{l} EB\_command, EB\_applied, c, time := \\ \quad FALSE, FALSE, 0, 0 \end{array}\right] \left(time \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times c\right) \\
\equiv & 0 \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times 0 \qquad \text{simple substitutions} \\
\equiv & 0 \qquad \text{arithmetic}
\end{array}
$$

Thus the initialisation event establishes the initial lower bound of the expectation.

It is also required to show that the events respect the expectation. Only the *EB_Request* event is examined here because this is the most interesting event (the *Safe_Failure* event trivially respects the expectation). To show that the *EB_Request* event respects the expectation, the following needs to be proved:

$$Inv \leq [EB\_Request]\, Inv \ .$$

Consider the right hand side of this inequality:

$$
\begin{array}{ll}
& [EB\_Request]\, Inv \\
\equiv & \text{definition of } EB\_Request \\
& \left[\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ {}_p\oplus \\ EB\_command, EB\_applied := TRUE, TRUE \end{array}\right) \\ \| \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array}\right] Inv \\
\equiv & \text{parallel substitution with } {}_p\oplus \\
& \left[\begin{array}{l} \left(\begin{array}{l} EB\_command, c := TRUE, c+1 \\ \| \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array}\right) \\ {}_p\oplus \\ \left(\begin{array}{l} EB\_command, EB\_applied := TRUE, TRUE \\ \| \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array}\right) \end{array}\right] Inv \\
\equiv & \text{probabilistic choice substitution } {}_p\oplus \text{ and definition of } Inv
\end{array}
$$

$$
\begin{array}{l}
p \times \\
\quad \left[\begin{array}{l} EB\_command, c := TRUE, c+1 \\ \| \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array}\right] \left(time \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times c\right) \\
\\
+ (1-p) \times \\
\quad \left[\begin{array}{l} EB\_command, EB\_applied := TRUE, TRUE \\ \| \\ time := time + exp\,(\lambda_{req} + \lambda_{safe}) \end{array}\right] \left(time \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times c\right)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \text{exponential, parallel and simple substitution} \\
& \begin{array}{rcl} p & \times & \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times (c+1)\right) \\ + \quad (1-p) & \times & \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times c\right) \end{array}
\end{array}
$$

To maintain the expectation it is required that $Inv \leq [main]\, Inv$, thus (from the above substitutions) it is required that:

$$\begin{array}{ll} & \left(time \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times c\right) \leq \\ & \quad p \times \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times (c+1)\right) + \\ & \quad (1-p) \times \left(\left(time + \frac{1}{\lambda_{req}+\lambda_{safe}}\right) \times \lambda_{max} - \frac{\lambda_{req}}{\lambda_{req}+\lambda_{safe}} \times c\right) \\ \equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{arithmetic} \\ & p \times \lambda_{req} \quad \leq \quad \lambda_{max} \end{array}$$

This concludes the analysis of the second expectation for the second stochastic Event-B model, the inequality found above was discussed in Section 4.4.

Appendix C

Proving the Healthiness Conditions in Deterministic sGCL

This appendix provides further details of the proof work required to show the healthiness conditions in the deterministic version of sGCL. In particular, this appendix provides the full proofs of continuity and linearity as these were only summarised in Section 6.3.1.

C.1 Continuity

The continuity of each of the program constructs is demonstrated. In some cases these are trivially continuous, but others require a more complex argument. Recall (Definition 6.3) that continuity is defined as follows:

Definition. An expectation transformer *wp.prog* is *boundedly continuous* iff the following holds

$$wp.prog.\,(\sqcup\mathcal{B}) \equiv (\sqcup Q : \mathcal{B} \cdot wp.prog.Q) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists. □

Abortion is trivially continuous as shown below:

Lemma C.1. The program abort is continuous, i.e.

$$wp.\mathsf{abort}.\,(\sqcup\mathcal{B}) \equiv (\sqcup Q : \mathcal{B} \cdot wp.\mathsf{abort}.Q) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists.

Proof.

$$\begin{array}{lll} & wp.\mathsf{abort}.\,(\sqcup\mathcal{B}) & \\ \equiv & 0 & wp \text{ definition} \\ \equiv & \sqcup\{0, 0, \ldots\} & \text{maximum of zeroes is zero} \end{array}$$

$\equiv \quad (\sqcup Q : \mathcal{B} \cdot wp.\mathsf{abort}.Q)$ *wp* definition

□

Identity is also trivially continuous as shown below:

Lemma C.2. The program skip is continuous, i.e.

$$wp.\mathsf{skip}.\,(\sqcup \mathcal{B}) \equiv (\sqcup Q : \mathcal{B} \cdot wp.\mathsf{skip}.Q) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists.

Proof.

$wp.\mathsf{skip}.\,(\sqcup \mathcal{B})$

$\equiv \quad (\sqcup \mathcal{B})$ *wp* definition

$\equiv \quad (\sqcup Q : \mathcal{B} \cdot wp.\mathsf{skip}.Q)$ *wp* definition

□

Assignment is continuous because an assignment operation simply transforms the current state to any other valid state. As the ordering of the expectations is defined for any state, updating a state will not change the ordering of the expectations, and hence the supremum will not be changed under assignment. A similar argument follows for stochastic assignment, which is reinforced by the monotone convergence theorem [72] as follows:

Lemma C.3. The program $(x :\oplus \mu)$ is continuous, i.e.

$$wp.\,(x :\oplus \mu)\,.\,(\sqcup \mathcal{B}) \equiv (\sqcup Q : \mathcal{B} \cdot wp.\,(x :\oplus \mu)\,.Q) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists.

Proof.

$wp.\,(x :\oplus \mu)\,.\,(\sqcup \mathcal{B})$

$\equiv \quad \displaystyle\int_{\mu} (\sqcup \mathcal{B})$ *wp* definition

$\equiv \quad \left(\sqcup Q : \mathcal{B} \cdot \displaystyle\int_{\mu} Q\right)$ monotone convergence theorem

$\equiv \quad (\sqcup Q : \mathcal{B} \cdot wp.\,(x :\oplus \mu)\,.Q)$ *wp* definition

□

The sequential composition of two continuous programs is trivially continuous as follows:

Lemma C.4. The program $prog_1; prog_2$ is continuous when programs $prog_1$ and $prog_2$ are continuous, i.e.

$$wp.\left(prog_1; prog_2\right).\left(\sqcup\mathcal{B}\right) \equiv \left(\sqcup Q : \mathcal{B} \cdot wp.\left(prog_1; prog_2\right).Q\right) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists.

Proof.

$$\begin{array}{lll}
 & wp.\left(prog_1; prog_2\right).\left(\sqcup\mathcal{B}\right) & \\
\equiv & wp.prog_1.\left(wp.prog_2.\left(\sqcup\mathcal{B}\right)\right) & wp \text{ definition} \\
\equiv & wp.prog_1.\left(\sqcup Q : \mathcal{B} \cdot wp.prog_2.Q\right) & prog_2 \text{ continuous} \\
\equiv & \left(\sqcup Q : \mathcal{B} \cdot wp.prog_1.\left(wp.prog_2.Q\right)\right) & prog_1 \text{ continuous} \\
\equiv & \left(\sqcup Q : \mathcal{B} \cdot wp.\left(prog_1; prog_2\right).Q\right) & wp \text{ definition}
\end{array}$$

□

Proving the continuity of probabilistic choice (and thus conditional choice) is also fairly straightforward:

Lemma C.5. The program $prog_1 \ {}_p\!\oplus\ prog_2$ is continuous when programs $prog_1$ and $prog_2$ are continuous, i.e.

$$wp.\left(prog_1 \ {}_p\!\oplus\ prog_2\right).\left(\sqcup\mathcal{B}\right) \equiv \left(\sqcup Q : \mathcal{B} \cdot wp.\left(prog_1 \ {}_p\!\oplus\ prog_2\right).Q\right) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists.

Proof.

$$\begin{array}{lll}
 & wp.\left(prog_1 \ {}_p\!\oplus\ prog_2\right).\left(\sqcup\mathcal{B}\right) & \\
\equiv & p * wp.prog_1.\left(\sqcup\mathcal{B}\right) + (1-p) * wp.prog_2.\left(\sqcup\mathcal{B}\right) & wp \text{ definition} \\
\equiv & p * \left(\sqcup Q : \mathcal{B} \cdot wp.prog_1.Q\right) + (1-p) * wp.prog_2.\left(\sqcup\mathcal{B}\right) & prog_1 \text{ continuous} \\
\equiv & p * \left(\sqcup Q : \mathcal{B} \cdot wp.prog_1.Q\right) + (1-p) * \left(\sqcup Q : \mathcal{B} \cdot wp.prog_2.Q\right) & prog_2 \text{ continuous} \\
\equiv & \left(\sqcup Q : \mathcal{B} \cdot p * wp.prog_1.Q\right) + \left(\sqcup Q : \mathcal{B} \cdot (1-p) * wp.prog_2.Q\right) & p \text{ not dependent on } Q \\
\equiv & \left(\sqcup Q : \mathcal{B} \cdot p * wp.prog_1.Q + (1-p) * wp.prog_2.Q\right) & \text{addition is continuous} \\
\equiv & \left(\sqcup Q : \mathcal{B} \cdot wp.\left(prog_1 \ {}_p\!\oplus\ prog_2\right).Q\right) & wp \text{ definition}
\end{array}$$

□

For completeness the proof of the continuity of a finite while-loop (Lemma 6.4) is repeated here:

Lemma. A *finite* loop $\mathsf{do}\ G \rightarrow body\ \mathsf{od}$ is continuous when the program *body* is continuous, i.e.

$$wp.\left(\mathsf{do}\ G \rightarrow body\ \mathsf{od}\right).\left(\sqcup\mathcal{B}\right) \equiv \left(\sqcup Q : \mathcal{B} \cdot wp.\left(\mathsf{do}\ G \rightarrow body\ \mathsf{od}\right).Q\right) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists.

Proof. The proof follows from unfolding the definition of the loop, based on the fact that a finite loop terminates within n iterations for some finite n:

$$
\begin{array}{lll}
 & wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\,(\sqcup\mathcal{B}) & \\
\equiv & & \text{unfolding (up to } n \text{ iterations)} \\
 & \quad [\neg G] * (\sqcup\mathcal{B}) & \\
 & +\ \ [G] * wp.body.\,([\neg G] * (\sqcup\mathcal{B})) & \\
 & +\ \ [G] * wp.body.\,([G] * wp.body.\,([\neg G] * (\sqcup\mathcal{B}))) & \\
 & +\ \ \ldots & \\
 & +\ \ ([G] * wp.body)^n\,.\,([\neg G] * (\sqcup\mathcal{B})) & \\
\equiv & & G \text{ not dependent on } Q \\
 & \quad (\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
 & +\ \ [G] * wp.body.\,(\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
 & +\ \ [G] * wp.body.\,([G] * wp.body.\,(\sqcup Q : \mathcal{B} \cdot [\neg G] * Q)) & \\
 & +\ \ \ldots & \\
 & +\ \ ([G] * wp.body)^n\,.\,(\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
\equiv & & \text{} body \text{ and sequential composition continuous, } G \text{ not dependent on } Q \\
 & \quad (\sqcup Q : \mathcal{B} \cdot [\neg G] * Q) & \\
 & +\ \ (\sqcup Q : \mathcal{B} \cdot [G] * wp.body.\,([\neg G] * Q)) & \\
 & +\ \ (\sqcup Q : \mathcal{B} \cdot [G] * wp.body.\,([G] * wp.body.\,([\neg G] * Q))) & \\
 & +\ \ \ldots & \\
 & +\ \ (\sqcup Q : \mathcal{B} \cdot ([G] * wp.body)^n\,.\,([\neg G] * Q)) & \\
\equiv & & \text{addition is continuous} \\
 & \left(\sqcup Q : \mathcal{B} \cdot \left(\begin{array}{ll} & [\neg G] * Q \\ +\ & [G] * wp.body.\,([\neg G] * Q) \\ +\ & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * Q)) \\ +\ & \ldots \\ +\ & ([G] * wp.body)^n\,.\,([\neg G] * Q) \end{array}\right)\right) & \\
\equiv & (\sqcup Q : \mathcal{B} \cdot wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.Q) & \text{folding (up to } n \text{ iterations)}
\end{array}
$$

where $([G] * wp.body)^n$ represents n nested applications of $[G] * wp.body$. For example, $([G] * wp.body)^2\,.X$ is interpreted as $[G] * wp.body.\,([G] * wp.body.X)$.

□

Recall (Section 6.2) that only finite while loops are allowed in sGCL, therefore this is sufficient.

C.2 Linearity

Each of the program constructs are considered individually to ensure that they are all linear. Most of them are fairly trivially linear, however recursion requires more thought. Recall (Section 6.3.1) that a transformer is linear iff

$$wp.prog.(a\alpha + b\beta) = a * wp.prog.\alpha + b * wp.prog.\beta \ ,$$

for expectations α, β and reals a, b.

Abortion is trivially linear as shown below:

Lemma C.6. The program abort is linear, i.e.

$$wp.\mathsf{abort}.\,(a\alpha + b\beta) = a * wp.\mathsf{abort}.\alpha + b * wp.\mathsf{abort}.\beta \ ,$$

for expectations α, β and reals a, b.

Proof.

$$\begin{array}{lll} & wp.\mathsf{abort}.\,(a\alpha + b\beta) & \\ \equiv & 0 & wp \text{ definition} \\ \equiv & 0 + 0 & \text{basic arithmetic} \\ \equiv & a * wp.\mathsf{abort}.\alpha + b * wp.\mathsf{abort}.\beta & \text{basic arithmetic, } wp \text{ definition} \end{array}$$

□

Identity is also trivially linear as shown below:

Lemma C.7. The program skip is linear, i.e.

$$wp.\mathsf{skip}.\,(a\alpha + b\beta) = a * wp.\mathsf{skip}.\alpha + b * wp.\mathsf{skip}.\beta \ ,$$

for expectations α, β and reals a, b.

Proof.

$$\begin{array}{lll} & wp.\mathsf{skip}.\,(a\alpha + b\beta) & \\ \equiv & a\alpha + b\beta & wp \text{ definition} \\ \equiv & a * wp.\mathsf{skip}.\alpha + b * wp.\mathsf{skip}.\beta & wp \text{ definition} \end{array}$$

□

Assignment according to some expression is linear as shown below:

Lemma C.8. The program $(x := E)$ is linear, i.e.

$$wp.\,(x := E)\,.\,(a\alpha + b\beta) = a * wp.\,(x := E)\,.\alpha + b * wp.\,(x := E)\,.\beta \ ,$$

for expectations α, β and reals a, b.

Proof.

$$\begin{array}{lll} & wp.\,(x := E)\,.\,(a\alpha + b\beta) & \\ \equiv & (a\alpha + b\beta)\,[x \backslash E] & wp \text{ definition} \\ \equiv & (a\alpha)\,[x \backslash E] + (b\beta)\,[x \backslash E] & \text{substitution distributes over addition} \end{array}$$

$$\begin{aligned}
&\equiv && a\,(\alpha[x\backslash E]) + b\,(\beta[x\backslash E]) && \text{no free } x \text{ in constants } a, b\\
&\equiv && a * wp.\,(x := E)\,.\alpha + b * wp.\,(x := E)\,.\beta && wp \text{ definition}
\end{aligned}$$

□

Assignment according to a continuous probability measure is linear as shown below:

Lemma C.9. The program $(x :\oplus \mu)$ is linear, i.e.

$$wp.\,(x :\oplus \mu)\,.\,(a\alpha + b\beta) = a * wp.\,(x :\oplus \mu)\,.\alpha + b * wp.\,(x :\oplus \mu)\,.\beta\ ,$$

for expectations α, β and reals a, b.

Proof.

$$\begin{aligned}
&&& wp.\,(x :\oplus \mu)\,.\,(a\alpha + b\beta)\\
&\equiv && \int_{\mu} (a\alpha + b\beta) && wp \text{ definition}\\
&\equiv && a\int_{\mu} \alpha + b\int_{\mu} \beta && \text{linearity of Lebesgue integration}\\
&\equiv && a * wp.\,(x :\oplus \mu)\,.\alpha + b * wp.\,(x :\oplus \mu)\,.\beta && wp \text{ definition}
\end{aligned}$$

□

The sequential composition of two linear programs is linear as shown below:

Lemma C.10. The program $(prog_1; prog_2)$ is linear, i.e.

$$wp.\,(prog_1; prog_2)\,.\,(a\alpha + b\beta) = a * wp.\,(prog_1; prog_2)\,.\alpha + b * wp.\,(prog_1; prog_2)\,.\beta\ ,$$

for expectations α, β, reals a, b and linear programs $prog_1, prog_2$.

Proof.

$$\begin{aligned}
&&& wp.\,(prog_1; prog_2)\,.\,(a\alpha + b\beta)\\
&\equiv && wp.prog_1.\,(wp.prog_2.\,(a\alpha + b\beta)) && wp \text{ definition}\\
&\equiv && wp.prog_1.\,(a * wp.prog_2.\alpha + b * wp.prog_2.\beta) && prog_2 \text{ linear}\\
&\equiv && a * wp.prog_1.\,(wp.prog_2.\alpha) + b * wp.prog_1.\,(wp.prog_2.\beta) && prog_1 \text{ linear}\\
&\equiv && a * wp.\,(prog_1; prog_2)\,.\alpha + b * wp.\,(prog_1; prog_2)\,.\beta && wp \text{ definition}
\end{aligned}$$

□

The probabilistic choice of two linear programs is linear as shown below:

Lemma C.11. The program $(prog_1\ {}_p\!\oplus\ prog_2)$ is linear, i.e.

$$\begin{aligned}
wp.\,(prog_1\ {}_p\!\oplus\ prog_2)\,.\,(a\alpha + b\beta) = \quad & a * wp.\,(prog_1\ {}_p\!\oplus\ prog_2)\,.\alpha\\
+ \quad & b * wp.\,(prog_1\ {}_p\!\oplus\ prog_2)\,.\beta\ ,
\end{aligned}$$

for expectations α, β, reals a, b and linear programs $prog_1, prog_2$.

Proof.

$$
\begin{array}{lll}
 & wp.\,(prog_1\ {}_p\!\oplus\ prog_2)\,.\,(a\alpha + b\beta) & \\
\equiv & p * wp.prog_1.\,(a\alpha + b\beta) + (1-p) * wp.prog_2.\,(a\alpha + b\beta) & wp \text{ definition} \\
\equiv & ap * wp.prog_1.\alpha + bp * wp.prog_1.\beta + (1-p) * wp.prog_2.\,(a\alpha + b\beta) & prog_1 \text{ linear} \\
\equiv & & prog_2 \text{ linear} \\
 & ap * wp.prog_1.\alpha + bp * wp.prog_1.\beta + & \\
 & a\,(1-p) * wp.prog_2.\alpha + b\,(1-p) * wp.prog_2.\beta & \\
\equiv & & \text{simple algebra} \\
 & a\,(p * wp.prog_1.\alpha + (1-p) * wp.prog_2.\alpha) + & \\
 & b\,(p * wp.prog_1.\beta + (1-p) * wp.prog_2.\beta) & \\
\equiv & a * wp.\,(prog_1\ {}_p\!\oplus\ prog_2)\,.\alpha + b * wp.\,(prog_1; prog_2)\,.\beta & wp \text{ definition}
\end{array}
$$

□

The conditional choice of two linear programs is also linear. The reasoning follows from that of probabilistic choice as conditional choice can be thought of as a special case of probabilistic choice where $p = [G]$ for some measurable set G.

For completeness the proof that a finite while-loop is linear (Lemma 6.5) is repeated here:

Lemma. A *finite* loop $\mathsf{do}\ G \to body\ \mathsf{od}$ is linear when the program *body* is linear, i.e.

$$
\begin{array}{rll}
wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\,(a\alpha + b\beta) = & & a * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\alpha \\
 & + & b * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\beta\ ,
\end{array}
$$

for expectations α, β, reals a, b and linear programs $prog_1, prog_2$.

Proof. The proof follows from unfolding the definition of the loop, based on the fact that a finite loop terminates within n iterations for some finite n:

$$
\begin{array}{lll}
 & wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\,(a\alpha + b\beta) & \\
\equiv & & \text{unfolding (up to } n \text{ iterations)} \\
 & [\neg G] * (a\alpha + b\beta) & \\
+ & [G] * wp.body.\,([\neg G] * (a\alpha + b\beta)) & \\
+ & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * (a\alpha + b\beta))) & \\
+ & \ldots & \\
+ & ([G] * wp.body)^n\,.\,([\neg G] * (a\alpha + b\beta)) &
\end{array}
$$

$\equiv$ simple algebra

$$
\begin{array}{lll}
 & a * [\neg G] * \alpha + b * [\neg G] * \beta \\
+ & [G] * wp.body.\,(a * [\neg G] * \alpha + b * [\neg G] * \beta) \\
+ & [G] * wp.body.\,([G] * wp.body.\,(a * [\neg G] * \alpha + b * [\neg G] * \beta)) \\
+ & \ldots \\
+ & ([G] * wp.body)^n\,.\,(a * [\neg G] * \alpha + b * [\neg G] * \beta)
\end{array}
$$

$\equiv$ *body* and sequential composition linear

$$
\begin{array}{lll}
 & a * [\neg G] * \alpha \\
+ & b * [\neg G] * \beta \\
+ & a * [G] * wp.body.\,([\neg G] * \alpha) \\
+ & b * [G] * wp.body.\,([\neg G] * \beta) \\
+ & a * [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \alpha)) \\
+ & b * [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \beta)) \\
+ & \ldots \\
+ & a * ([G] * wp.body)^n\,.\,([\neg G] * \alpha) \\
+ & b * ([G] * wp.body)^n\,.\,([\neg G] * \beta)
\end{array}
$$

$\equiv$ simple algebra

$$
\begin{array}{lll}
 & & [\neg G] * \alpha \\
 & + & [G] * wp.body.\,([\neg G] * \alpha) \\
a * & + & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \alpha)) \\
 & + & \ldots \\
 & + & ([G] * wp.body)^n\,.\,([\neg G] * \alpha) \\
 & & [\neg G] * \beta \\
 & + & [G] * wp.body.\,([\neg G] * \beta) \\
+\ b * & + & [G] * wp.body.\,([G] * wp.body.\,([\neg G] * \beta)) \\
 & + & \ldots \\
 & + & ([G] * wp.body)^n\,.\,([\neg G] * \beta)
\end{array}
$$

$\equiv$ folding (up to n iterations)

$$a * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\alpha + b * wp.\,(\mathsf{do}\ G \to body\ \mathsf{od})\,.\beta$$

where $([G] * wp.body)^n$ represents n nested applications of $[G] * wp.body.$ For example, $([G] * wp.body)^2\,.X$ is interpreted as $[G] * wp.body.\,([G] * wp.body.X)$.

□

As with continuity, this is sufficient because recursion is restricted to finite while loops in sGCL.

Appendix D

The Challenge of Proving Consistency in a Non-Deterministic sGCL

This appendix discusses in detail the issues in showing the consistency between the transformer and relational semantics of a non-deterministic sGCL as described in Chapter 7. It also includes an aside on the use of the Kantorovich metric with sub-probability measures.

D.1 Consistency Proofs

In this section the challenge of proving the consistency of the relational and transformer semantics is discussed. This involves two main steps: showing a $1-1$ correspondence between the two semantics through mutual inverses wp and rp (Definitions 7.16 and 7.18); and showing that all of the operations have equivalent definitions in both semantics. The first of these is discussed in detail in this section and is split into two sub-tasks: that of showing that wp is an injection, and likewise for rp. The second step has not been explored in depth, as these proofs are dependant on how the issues arising from the first step are resolved.

When considering the proofs that wp and rp are both injections, the approach examined is to approximate the continuous probability measures with appropriate discrete equivalents, to enable re-use of the proofs provided for pGCL where possible. Therefore, such an approximation (along with some useful properties of it) are described before discussing the injection proofs.

Note that whilst some of the definitions and lemmas given here are only shown for a one-dimensional state space, it is believed that the results will also hold for an n-dimensional state space (with n finite).

D.1.1 A discrete approximation for measures

A discrete approximation for measures is described, and some useful properties of it are proved. It is intended that this provides the building blocks for proving that *wp* and *rp* are injections. Note that (with the intention of wider applicability) these proofs have been shown for the entire real space where possible. It is trivial to specialise them to closed intervals of the form $[a, b]$ as required for the injection proofs.

The ϵ-approximation of a measure μ is defined as follows:

Definition D.1. An *ϵ-approximation*, μ_ϵ, of μ is defined for any $\epsilon > 0$ as:

$$\mu_\epsilon := \sum_i a_i * \overline{x_i} \ ,$$

where $i \in \mathbb{Z}$, and for every interval

$$A_i := [i * \epsilon, (i+1)\,\epsilon] \ ,$$

$\overline{x_i}$ represents the point distribution at its midpoint $\left(i + \frac{1}{2}\right)\epsilon$ and $a_i = \mu.A_i$ is a weight according to μ. □

The ϵ-approximation essentially divides the state space of the measure into intervals of width ϵ and provides a discrete approximation for each using a point distribution that is weighted according to the original measure.

Figure D.1 illustrates the ϵ-approximation for a truncated[1] exponential distribution, where ϵ takes the value 0.2 in this case. Note that the illustration is given for the distribution function. This is because the density function of a point distribution is infinitesimally narrow and tall, and therefore difficult to plot.

The measure and its approximation can be arbitrarily close

The first and crucial property of the ϵ-approximation is that ϵ can be chosen so that the approximation is arbitrarily close (according to the Kantorovich metric) to the original measure. More precisely it is shown that an ϵ-approximation μ_ϵ is within ϵ distance (under the Kantorovich metric) of the original measure μ. This is shown in Lemma D.2:

Lemma D.2. The *ϵ-approximation* of a measure μ is within ϵ of μ (according to the Kantorovich metric), i.e.

$$K.\mu.\mu_\epsilon \leq \epsilon$$

where μ_ϵ is as defined in Definition D.1.

Proof. Consider a single interval A_i as defined in Definition D.1, the greatest possible (Kantorovich) distance between μ and μ_ϵ needs to be determined over interval A_i. The

[1] The truncated exponential distribution is simply the exponential distribution cut off at some finite value and re-scaled so that the total probability defined is still one.

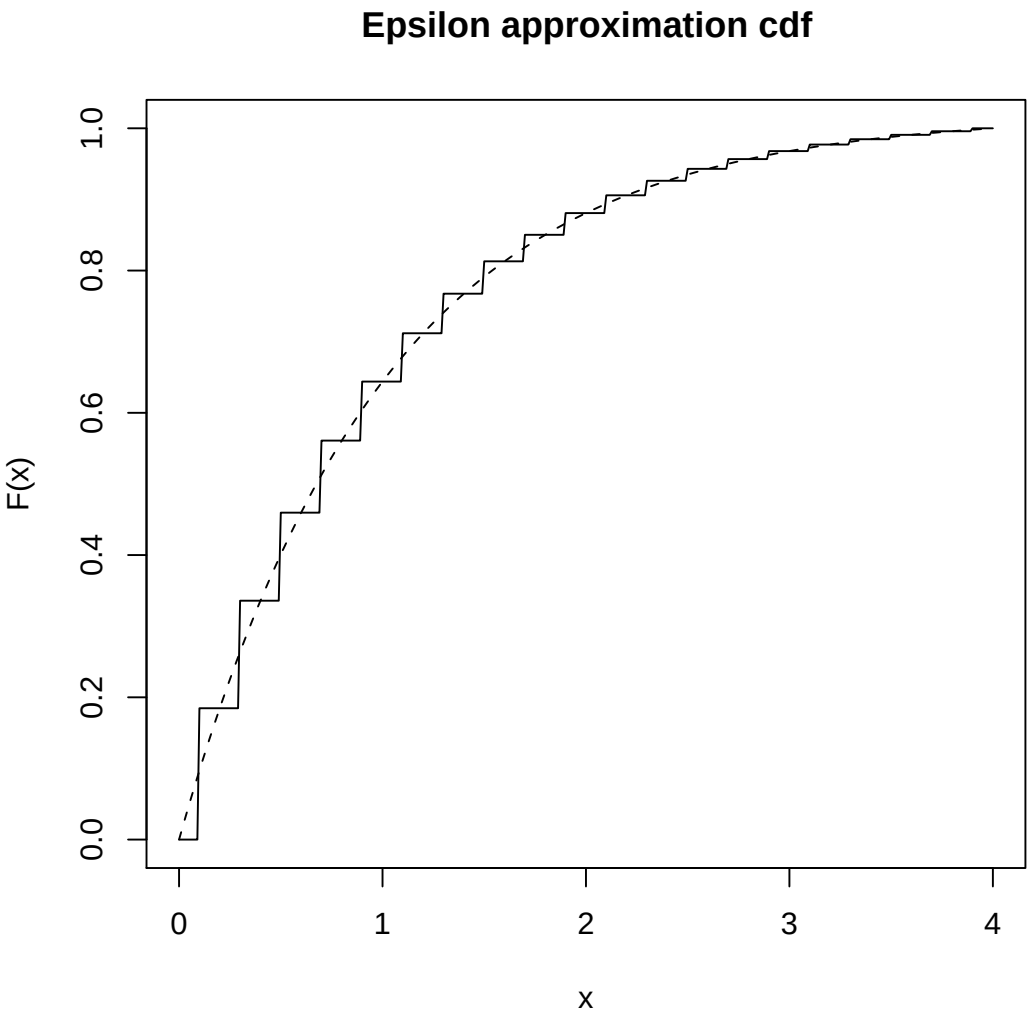

Figure D.1: The ϵ-approximation (solid) of a truncated exponential distribution (dashed)

Kantorovich distance between μ and μ_ϵ is by definition the area enclosed between the curves of their respective distribution functions[2]. For any two arbitrary measures μ and μ_ϵ the maximal area between the two curves is simply the width of the interval, ϵ, multiplied by the total probability within that interval, $\mu.A_i$. However as μ_ϵ is defined to be the point distribution at the mid-point of the interval (which bisects the space) and both μ and μ_ϵ must be monotonically increasing, the maximal area between the two graphs is halved giving $\frac{\epsilon}{2} * \mu.A_i$.

Now find the maximal Kantorovich distance between arbitrary μ and μ_ϵ by summing the maximal distances of all of the partitions:

[2]Note that this use of the definition of the Kantorovich metric for real numbers is valid here. This is because, by definition, the total probability defined for a measure and its ϵ-approximation will always be the same.

$$
\begin{array}{lll}
 & \sup\ \{K.\mu.\mu_\epsilon \mid \mu \in \bar{S}\} & \\
\equiv & \displaystyle\sum_{i\in\mathbb{Z}} \frac{\epsilon}{2} * \mu.A_i & \text{by definition of } \mu_\epsilon \text{ and explanation above} \\
\equiv & \displaystyle\frac{\epsilon}{2} \sum_{i\in\mathbb{Z}} \mu.A_i & \tfrac{\epsilon}{2} \text{ not dependant on } i \\
\leq & \displaystyle\frac{\epsilon}{2} & \text{distinct partitions and } \mu.[0,1] \leq 1 \\
< & \epsilon & \text{simple algebra}
\end{array}
$$

□

It is interesting to note that this approximation result applies to the infinite domain because the total measure of the entire state space must always be no more than one, no matter how big the domain is.

Integration over synchronised simple functions are equal

The next property of interest states that the Lebesgue integral of certain functions are equivalent with respect to the measure and its ϵ-approximation. More specifically, for some simple function f (see Section 5.1.2) whose constituent indicator functions operate over the intervals A_i of the approximation μ_ϵ, the integral of f over μ_ϵ is the same as that over μ. This is shown in Lemma D.3.

Lemma D.3. For ϵ-approximation μ_ϵ of a probability measure μ,

$$\int_S f \ \mathrm{d}\mu = \int_S f \ \mathrm{d}\mu_\epsilon \ ,$$

where $f.x = \sum_i a_i I_{A_i}.x$ is a simple function with reals a_i, indicator functions $I_{A_i}.x$, and μ_ϵ, A_i are as given in Definition D.1.

Proof. This is proved by showing that the difference between these two integrals is zero:

$$
\begin{array}{lll}
 & \displaystyle\int_S f \ \mathrm{d}\mu - \int_S f \ \mathrm{d}\mu_\epsilon & \\
\equiv & \displaystyle\sum_i a_i \mu.A_i - \sum_i a_i \mu_\epsilon.A_i & \text{Definition 5.4, } A_i\text{'s disjoint} \\
\equiv & \displaystyle\sum_i a_i \mu.A_i - \sum_i a_i \mu.A_i * \overline{x_i}.A_i & \text{Definition D.1} \\
\equiv & \displaystyle\sum_i a_i \mu.A_i - \sum_i a_i \mu.A_i * 1 & \text{by definition of } \bar{s} \text{ (Definition 7.3), } x_i \in A_i \\
\equiv & 0 & \text{simple arithmetic}
\end{array}
$$

□

This property allows the ϵ-approximation to be used in situations where the equality of the integrals of the measures over these kinds of simple functions is important, as opposed to the equality of the measures themselves.

The distance between two approximations is bounded

This property shows that for two measures, μ and μ', that are separated by a Kantorovich distance of d, the distance between their respective approximations μ_ϵ and μ'_ϵ is bounded below by $d-2\epsilon$. A similar argument can also be used to show that the approximations are bounded above by $d+2\epsilon$. Intuitively, this is the first step towards showing that a distance between two measures leads to a distance between their respective approximations. The lower bound of two approximations is proved in Lemma D.4.

Lemma D.4. For two measures μ and μ' with Kantorovich distance $K.\mu.\mu' = d$,

$$K.\mu_\epsilon.\mu'_\epsilon \geq d - 2\epsilon \ ,$$

where μ_ϵ is as defined in Lemma D.2.

Proof. This is proved by using the (metric space) triangle inequality $d.x.z \leq d.x.y+d.y.z$:

$$\begin{array}{lll} & K.\mu_\epsilon.\mu'_\epsilon & \\ \geq & K.\mu.\mu'. - (K.\mu.\mu_\epsilon + K.\mu'.\mu'_\epsilon) & \text{triangle inequality and simple algebra} \\ \geq & d - 2\epsilon & K.\mu.\mu' = d \text{ and Lemma D.2} \end{array}$$

□

Corollary. Using similar reasoning the following also holds:

$$K.\mu_\epsilon.\mu'_\epsilon \leq d + 2\epsilon$$

□

Note that if ϵ is chosen to be $\frac{d}{3}$, it trivially follows that $K.\mu_\epsilon.\mu'_\epsilon > 0$, for all $d > 0$. Also, if $d < 2\epsilon$ then $K.\mu_\epsilon.\mu'_\epsilon \geq 0$, as distance cannot be negative.

Compactness is preserved by the closure of a set of approximations

Compactness is an important property for proving the relationship between the transformer and relational semantics. This next property shows that taking the Kantorovich-closure of a set of ϵ-approximations of a compact set of measures results in a compact set. However, this proof only holds when the state space is restricted to measures over closed intervals $[a, b]$, because the set of all measures over any closed interval $[a, b]$ is compact [82]. This is shown in Lemma D.5.

Lemma D.5. For a compact set of measures C over a closed interval $[a, b]$ for reals a, b, the closure of the set of ϵ-approximations, C_ϵ, of measures in C is also compact. Here the ϵ-approximation of a measure is as given in Definition D.1 and compactness is defined according to the Kantorovich metric.

Proof. The proof uses the fact that a closed subset of a compact set is also compact [77, Section 1.3]. Consider the set of all measures over $[a, b]$ and call it KS. This is compact under the Kantorovich metric because the interval $[a, b]$ is compact [82]. Both C and C_ϵ are subsets of KS as the state space of the measures in C is restricted to $[a, b]$, and the ϵ-approximation of a measure does not change the values it ranges over. Examining C_ϵ the following holds (where $Cl.X$ denotes the closure of set X) :

$$
\begin{array}{lll}
 & C_\epsilon \subseteq KS & \\
\implies & Cl.C_\epsilon \subseteq Cl.KS & \text{closing both sides} \\
\implies & Cl.C_\epsilon \subseteq KS & KS \text{ is compact and thus closed}
\end{array}
$$

Therefore the closure of C_ϵ is a subset of a compact set and as such is itself compact. □

A difference between measures can be detected by its approximations

The final property that is shown for the approximations is that if there is some separation between a compact set of measures C and a measure μ, this difference can be detected by their respective ϵ-approximations, for some appropriate choice of ϵ. Specifically, that for some set of measures C and a measure μ not in C, there exists some $\epsilon > 0$ such that μ_ϵ is not in the closure of C_ϵ. This is shown in Lemma D.6.

Lemma D.6. For a compact set of measures C and a measure $\mu \notin C$,

$$\exists \epsilon \in \mathbb{R}^+ \cdot \mu_\epsilon \notin Cl.C_\epsilon \ ,$$

where $Cl.X$ denotes the closure of X and the ϵ-approximation of a measure is as given in Definition D.1.

Proof. If $\mu \notin C$ then $\forall c \in C \cdot K.\mu.c > 0$ as $K.x.y = 0 \implies x = y$. Define $K.\mu.C$ to be $\inf \{K.\mu.c \mid c \in C\}$, the distance between C and μ, and c' to be the measure in C such that $K.\mu.c' = K.\mu.C$. Assume $K.\mu.C = d$ for some $d \in \mathbb{R}^+$ (from the first statement d must be greater than zero). By Lemma D.4 $K.\mu_\epsilon.c'_\epsilon \geq d - 2\epsilon$, therefore select $\epsilon = \frac{d}{3}$ to be a witness so that $K.\mu_\epsilon.c'_\epsilon \geq d - \frac{2}{3}d > 0$ as $d > 0$. As c' is the closest measure to μ, the distance between all other $c \in C$ must be strictly greater than $d - \frac{2}{3}d$ and thus also greater than zero. Therefore, $\epsilon = \frac{d}{3} \implies K.\mu_\epsilon.Cl.C_\epsilon > 0 \implies \mu_\epsilon \notin Cl.C_\epsilon$. Note that this also follows for any $\epsilon < \frac{d}{2}$. □

D.1.2 The challenge of showing that wp is an injection

Showing that the relational to transformer retraction wp is an injection is the first step in demonstrating a 1–1 relationship between the two semantics. However, the proof is particularly challenging. The existence of a separating hyperplane between a compact set of measures and a further distinct measure needs to be shown. In pGCL the use of

a geometric result, called the separating hyperplane lemma (Lemma E.1), can be used almost directly by converting discrete distributions into Euclidean space. Unfortunately, continuous measures can not be represented in Euclidean space in the same way, so more work is required to make use of the geometric result.

Two proof strategies are explored below. The first strategy gets close to a complete proof of the *wp*-injection, but is blocked by an issue with transferring the compactness arguments from the continuous space to the discrete approximation in Euclidean space. The second strategy looks like a promising resolution to this problem but requires further work to prove all of the lemmas.

First proof strategy

It is proposed to approximate continuous probability measures using the ϵ-approximation described above (Section D.1.1). Using this approach, the proof that wp is an injection is outlined below (Lemma D.7), before discussing the issues that occur. The proof that wp is an injection is based on that given by McIver et al. for pGCL [61, Lemma 5.7.2], and takes the contrapositive as outlined below:

Lemma D.7. For any $r \in \mathbb{H}S$,

$$rp.\,(wp.r) = r\ .$$

Proof. For arbitrary $\mu \in \bar{S}$:

$$\begin{array}{lll}
 & \mu \notin rp.\,(wp.r)\,.s & \\
\text{iff} & \mu \notin \left\{\mu : \bar{S} \mid \left(\forall \beta : \mathbb{E}S \cdot wp.r.\beta.s \leq \int_\mu \beta\right)\right\} & \text{Definition 7.18} \\
\text{iff} & \left(\exists \beta : \mathbb{E}S \cdot wp.r.\beta.s > \int_\mu \beta\right) & \text{simple logic} \\
\text{iff} & \left(\exists \beta : \mathbb{E}S \cdot \left(\sqcap \mu' : r.s \cdot \int_{\mu'} \beta\right) > \int_\mu \beta\right) & \text{Definition 7.16} \\
\text{iff} & \left(\exists \beta : \mathbb{E}S, c : \mathbb{R} \cdot \left(\forall \mu' : r.s \cdot \int_{\mu'} \beta > c\right) \wedge \int_\mu \beta < c\right) & \text{min } A > x \text{ implies all } a \in A > x \\
\text{iff} & \mu \notin r.s & \dagger
\end{array}$$

The justification of the *if* indicated by † can be shown using the separating hyperplane lemma (Lemma E.1). This requires the conversion of measures into Euclidean space. Then the compact set $\mathcal{C}$ is taken to be $r.s$, the point p is simply μ and the separating hyperplane S obtained provides the required β.

For continuous probability measures the conversion to Euclidean space is achieved by taking the ϵ-approximations of the measures involved. Then each point distribution is considered to be an axis, with its weight as the coordinate for that axis. The value of ϵ is chosen so that separation of the measures can be detected by their ϵ-approximations. Define the (Kantorovich) distance between μ and $r.s$ as d[3]. Set ϵ to $\frac{d}{3}$, this ensures that $\mu_\epsilon \notin Cl.(r.s)_\epsilon$ by Lemma D.6.

[3]Note that the distance d must be greater than zero because $\mu \notin r.s$.

The separating hyperplane lemma can then be applied to the set $Cl.(r.s)_\epsilon$ with point μ_ϵ instead. However, there are still two issues to resolve before this can be achieved. The first of these is that the separating hyperplane lemma provides a set of coefficients a_i and a constant c that describe a plane S, not an expectation. However, these coefficients can be used to create the simple function $\beta = \sum_i a_i I_{A_i}$ where the A_i's are the corresponding intervals of the ϵ-approximation representing the axes in Euclidean space. The c goes into the equation directly. The integral of the measures with respect to such a β is equivalent to the integral of their ϵ-approximations with respect to β by Lemma D.3. This means that the β found for the ϵ-approximations applies to the original measures as well.

The second issue is unfortunately more challenging. The separating hyperplane lemma requires a *compact* set of points. If the measures are only defined over compact intervals, it can be shown that $Cl.(r.s)_\epsilon$ is compact *with respect to the Kantorovich metric* by Lemma D.5. However, once the approximations are converted to Euclidean space it is required to show that this representation is compact *with respect to the Euclidean metric*. This is not a trivial result to show, and is explored in more detail below.

□

The main issue in proving that *wp* is an injection is showing that for a set of ϵ-approximations, compactness according to the Kantorovich metric implies compactness according to the Euclidean metric. It is fairly trivial to show that a limit of a set of ϵ-approximations according to the Euclidean space metric is also a limit according to the Kantorovich metric. This is outlined below in Lemma D.8.

Lemma D.8. A Euclidean limit of a set of ϵ-approximations C_ϵ represented in Euclidean space (as described in Lemma D.7) is also a Kantorovich limit of C_ϵ.

Proof. An ϵ-approximation μ_ϵ of a measure μ is of the form

$$\sum_i a_i * \overline{x_i} \ ,$$

In Euclidean space, each $\overline{x_i}$ represents an axis and the a_i value is the coordinate for that axis. To find the Euclidean limit of a set of ϵ-approximations requires finding the limit of each of the a_i's. Therefore, define a_i^* as the limit of each sequence of a_i's, $\{a_{i1}, a_{i2}, ..., \}$. The limits are then put back into an ϵ-approximation to give ν as follows

$$\nu = \sum_i a_i^* * \overline{x_i} \ .$$

Define also a Kantorovich limit μ of C_ϵ that is of the same form as an ϵ-approximation,

$$\mu = \sum_i a_{i\mu} * \overline{x_i} \ .$$

The Kantorovich distance between μ and ν (assuming that the values of the x_i's and the total of the weights for each are the same) is

$$K.\mu.\nu = \sum_{i \geq 2} \left((x_i - x_{i-1}) \left| \sum_{j=1}^{i-1} a_{j\mu} - a_j^* \right| \right) = \epsilon \sum_{i \geq 2} \left(\left| \sum_{j=1}^{i-1} a_{j\mu} - a_j^* \right| \right) \ ,$$

which is exactly zero when $a_{i\mu} = a_i^*$ for all i, i.e. $\mu = \nu$ when the weights are identical. Therefore a Euclidean limit of a set of ϵ-approximations is also a Kantorovich limit.

□

However, the inverse is much more challenging to show, the problem is being sure that the closure of a set of ϵ-approximations only adds measures where the weights of the point distributions differ, not their positions.

One idea that was explored for inferring closure under the Euclidean metric from closure under the Kantorovich metric was to go via an intermediate metric space. It was intended that the intermediate metric space could be described in a similar way to the Kantorovich metric, but also be closely related to the Euclidean metric space. The "maximum" metric[4] looks like a promising intermediate metric space at first glance. It is defined as the maximum difference along any coordinate dimension [15, Chapter 10]:

Definition D.9. The *maximum* distance between two points x and y is defined as

$$M.x.y := \max_i \left(|x_i - y_i| \right) \ ,$$

where x_i, y_i are the i-coordinates of x and y respectively. □

The maximum distance can be defined between two ϵ-approximations as follows:

Definition D.10. The maximum distance between two ϵ-approximations μ_ϵ and μ'_ϵ is defined as

$$M.\mu_\epsilon.\mu'_\epsilon := \sup \left\{ \left| \int_{\mu_\epsilon} g - \int_{\mu'_\epsilon} g \right| \cdot g = \{[x_i]\} \right\} \ ,$$

where x_i represents the location of the point distributions in the ϵ-approximations. □

Note the similarity between Definitions 5.11 and D.10. The body of the expression is identical, only the restriction differs. Therefore if g can be shown to be a subset of f, it can be observed that $M.\mu_\epsilon.\mu'_\epsilon \leq K.\mu_\epsilon.\mu'_\epsilon$ for any $\mu_\epsilon, \mu'_\epsilon$. Such a relationship would enable the conclusion that limits under the Kantorovich metric are also limits under the maximum metric. Unfortunately, the sets of functions f and g are mutually exclusive. The set g only includes indicator functions, whilst the set f only includes 1-Lipschitz

[4] Also known as the Chebyshev metric, the L_∞ metric and the chessboard metric. The latter of these names comes from the fact that in two-dimensional space the maximum distance is the same as calculating the minimum number of moves required for a king in the game of chess to move from one square to another.

functions. The 1-Lipschitz functions by definition have a maximum gradient everywhere of one, but indicator functions have infinite gradient in places. Therefore, there is not even any overlap between these two sets.

A potential resolution to this problem is discussed in the second proof strategy below. This uses an alternative metric space to bridge the gap between the Kantorovich and Euclidean metrics. It looks like a promising direction, but further work is required to prove all of the required lemmas it generates.

The second proof strategy

The second strategy considered for the *wp* injection proof uses the Manhattan metric as a bridge between the Kantorovich and Euclidean metric spaces, instead of the maximum metric [64]. It also requires a slightly different basis for the state space of measures than was given in Chapter 6 and a further version of the Kantorovich metric. This section describes the lemmas that need to be proved for this strategy to work. The full proof of these lemmas is beyond the scope of this thesis.

First the state space S needs to be redefined. The new state space (call it S') is still a metric space containing closed intervals $[a, b]$ for reals a and b. However, the distance is no longer standard Euclidean distance. A scaled Euclidean distance,

$$d.x.y := \frac{|x - y|}{b - a} \ ,$$

is used instead, making the metric space one-bounded. This means that the Kantorovich metric over the space S' also has the range $[0, 1]$. Note that this state space change has a minimal impact on the overall semantics of sGCL[5].

The definition of the Kantorovich metric over one-bounded metric spaces can be specialised [82] as follows:

Definition D.11. Given any two Borel probability measures μ and ν on a one-bounded, metric space (S', d), the *Kantorovich distance* between μ and ν is defined by

$$K.\mu.\nu := \sup \left\{ \left| \int_{S'} f \ \mathrm{d}\mu - \int_{S'} f \ \mathrm{d}\nu \right| \cdot f \in S' \underset{1}{\to} [0, 1] \right\} \ ,$$

where $f \in S' \underset{1}{\to} [0, 1]$ denotes the set of 1-Lipschitz functions $S' \to [0, 1]$ so that $\forall x, y \cdot |f(x) - f(y)| \leq d(x, y)$. □

Based on Definition D.11, define KS' to be the metric space of Borel sub-probability measures on S' with the Kantorovich metric. This space is by definition compact and one-bounded [82].

The ϵ-approximation of a measure μ as given in Definition D.1 may be written either as μ_ϵ or $approx.\epsilon.\mu$ where appropriate. Recall that this provides a means of describing

[5] Only the scaling of the Kantorovich metric would be affected in the non-deterministic version of sGCL.

a measure as a point in Euclidean space where each approximation interval is an axis and the measure of that interval is the coordinate value for that axis.

Finally the Manhattan metric between two ϵ-approximations is defined as follows:

Definition D.12. The *Manhattan* distance between two ϵ-approximations μ_ϵ and μ'_ϵ is defined as

$$Mh.\mu_\epsilon.\mu'_\epsilon := \sum_{x_i} \left| \mu_\epsilon.x_i - \mu'_\epsilon.x_i \right| \ ,$$

where x_i represents the location of the point distributions in the ϵ-approximations. □

The fact that the Manhattan metric is topologically equivalent to the Euclidean metric in the space of the ϵ-approximations [64] assists in the following lemmas. These aim to show that key properties are preserved when converting the measures from the Kantorovich metric space to the Euclidean metric space. The first (Lemma D.13) requires that non-emptiness, convexity and up-closure are preserved in the transformation between metric spaces:

Lemma D.13. If set Z of measures in KS' is non-empty, up-closed, convex and compact, then

$$Z_\epsilon := image.\left(approx.\epsilon\right).Z$$

is non-empty, convex and up-closed in Euclidean space, where

$$image.f.Z := \{f.z \cdot z : Z\}$$

for function f and set Z. □

The proof of this and the subsequent lemma are beyond the scope of this thesis. The second (Lemma D.14) requires that non-emptiness, convexity and up-closure are preserved by Manhattan-closure. This allows the Manhattan-closure to be taken of the Euclidean representation of the ϵ-approximations without losing these important properties.

Lemma D.14. The Manhattan-closure of a non-empty, convex and up-closed set of points in $[0,1]^{\frac{b-a}{\epsilon}}$ is also non-empty, convex and up-closed. □

Note that the space $[0,1]^{\frac{b-a}{\epsilon}}$ is the space of points of ϵ-approximations represented in Euclidean space.

Based on these two lemmas it can be shown that using the Manhattan-closure and ϵ-approximation preserves the key properties of sets of measures as follows:

Lemma D.15. If set Z of measures in KS' is non-empty, up-closed, convex and compact, and

$$Z_\epsilon := image.\left(approx.\epsilon\right).Z$$

where $image.f.Z := \{f.z \cdot z : Z\}$ for function f and set Z, then the Manhattan-closure

$$MhCl.Z_\epsilon$$

is non-empty, up-closed, convex and compact in the Euclidean topology.

Proof. Follows directly from Lemmas D.13 and D.14, and the explicit Manhattan-closure of Z_ϵ. □

Lemma D.15 resolves the closure issue that was encountered in the first proof strategy by delaying the direct closure until the measures are represented in Euclidean space. However, this means that the lemma stating the preservation of a detectable distance between measures through the ϵ-approximation (Lemma D.6) is no longer sufficient. A new lemma is required to deal with this issue as given below:

Lemma D.16. Let μ_ϵ and μ'_ϵ be two ϵ-approximations of measures $\mu, \mu' \in KS'$ respectively such that

$$K.\mu_\epsilon.\mu'_\epsilon \geq d \ ,$$

for some $d > 0$, then

$$Mh.\mu_\epsilon.\mu'_\epsilon \geq d$$

also holds.

Proof.

$$\begin{aligned}
& K.\mu_\epsilon.\mu'_\epsilon \\
\equiv \quad & \sup\left\{\left|\int_{S'} f \ \mathrm{d}\mu_\epsilon - \int_{S'} f \ \mathrm{d}\mu'_\epsilon\right| \cdot f \in S' \underset{1}{\to} [0,1]\right\} && \text{Definition D.11} \\
\equiv \quad & && \text{Lebesgue integration, Definition D.1} \\
& \sup\left\{\left|\sum_{x_i} \left(f.x_i * \left(\mu_\epsilon.x_i - \mu'_\epsilon x_i\right)\right)\right| \cdot f \in S' \underset{1}{\to} [0,1]\right\} \\
\leq \quad & \sup\left\{\sum_{x_i} \left|\mu_\epsilon.x_i - \mu'_\epsilon x_i\right| \cdot f \in S' \underset{1}{\to} [0,1]\right\} && \text{algebra, range of } f \text{ is } [0,1] \\
\equiv \quad & \sum_{x_i} \left|\mu_\epsilon.x_i - \mu'_\epsilon x_i\right| && \text{not dependent on } f \\
\equiv \quad & Mh.\mu_\epsilon.\mu'_\epsilon && \text{Definition D.12}
\end{aligned}$$

□

Corollary. For measure μ not in Z, a non-empty, up-closed, convex and compact set of measures in KS', if ϵ is chosen such that for some $d > 0$

$$\forall \mu'_\epsilon \in Z_\epsilon \cdot K.\mu_\epsilon.\mu'_\epsilon \geq d \ ,$$

where $Z_\epsilon = image.\left(approx.\epsilon\right).Z$ then

$$\forall \mu'_\epsilon \in Z_\epsilon \cdot Mh.\mu_\epsilon.\mu'_\epsilon \geq d \ ,$$

and so μ_ϵ can also not be a member of the Manhattan-closure $MhCl.Z_\epsilon$. This follows directly from the above proof. □

The above, Lemma D.16, shows that the distinction between measures is preserved in the conversion from the Kantorovich metric space to the Euclidean metric space via the ϵ-approximation, as long as a suitable choice of ϵ is chosen as per Lemma D.7. Using this and the compactness that follows from Lemma D.15, the *wp*-injection proof can be completed. However, Lemmas D.13 and D.14 have been left without proof. This is beyond the scope of the thesis, although it is anticipated that these can be proven and thus that the whole *wp*-injection proof can be discharged.

D.1.3 The challenge of showing that rp is an injection

The next proof to explore is the inverse relationship, that rp is also an injection. Due to the complexity of this proof, the two directions $wp.(rp.t) \sqsupseteq t$ and $wp.(rp.t) \sqsubseteq t$ are considered separately. The former of these can be trivially shown based on the equivalent proof in pGCL [61, Lemma 5.7.3] as follows:

Lemma D.17. For any $t \in \mathbb{T}S$, if $rp.t$ is defined then

$$wp.\left(rp.t\right) \sqsupseteq t \ .$$

Proof. This follows directly from Definitions 7.16 and 7.18 for any $\beta \in \mathbb{E}S$ and $s \in S$:

$$\begin{array}{lll}
 & wp.\left(rp.t\right).\beta.s \geq t.\beta.s & \\
\text{iff} & \left(\sqcap \mu : rp.t.s \cdot \int_\mu \beta\right) \geq t.\beta.s & \text{Definition 7.16, } rp.t \text{ defined} \\
\text{iff} & \left(\forall \mu : rp.t.s \cdot \int_\mu \beta \geq t.\beta.s\right) & \\
\text{iff} & \left(\forall \mu : \bar{S} \cdot \left(\forall \beta' : \mathbb{E}S \cdot t.\beta'.s \leq \int_\mu \beta'\right) \implies \int_\mu \beta \geq t.\beta.s\right) & \text{Definition 7.18} \\
\text{iff} & true &
\end{array}$$

□

However, the other direction is more complex and has to be restricted to the subset of transformers that are sublinear and continuous. Recall (Definition 7.1)that sublinearity is defined as:

Definition. An expectation transformer $t \in \mathbb{T}S$ is *sublinear* iff for all $\beta_1, \beta_2 \in \mathbb{E}S$ and $c, c_1, c_2 \in \mathbb{R}_\geq$ the following holds

$$c_1\left(t.\beta_1\right) + c_2\left(t.\beta_2\right) \ominus \underline{c} \leq t.\left(c_1\beta_1 + c_2\beta_2 \ominus \underline{c}\right) \ .$$

where $x \ominus y$ means $(x - y) \sqcup 0$. □

and continuity is defined (Definition 6.3) as:

Definition. An expectation transformer $t \in \mathbb{T}S$ is *boundedly continuous* iff the following holds

$$t.(\sqcup\mathcal{B}) \equiv (\sqcup\beta : \mathcal{B} \cdot t.\beta) \ ,$$

where $\mathcal{B}$ is a $\leq$-directed and bounded subset of $\mathbb{E}S$, such that $\sqcup\mathcal{B}$ exists. □

The proof that $wp.(rp.t) \sqsubseteq t$ in pGCL [61, Lemma 5.7.4] is based on another geometric result called Farkas' lemma (see Lemma E.3). Therefore the proposed approach to proving this in sGCL is to convert the continuous measures into discrete ones once more using the ϵ-approximation (Definition D.1) again. However, once again there are issues with showing compactness. The proof is outlined and discussed below in Lemma D.18.

Lemma D.18. If $t \in \mathbb{T}S$ is sublinear and continuous and $rp.t$ is defined, then

$$wp.\,(rp.t) \sqsubseteq t \ .$$

Proof. Using the contrapositive approach to this proof, suppose that $rp.t$ is defined but $wp.\,(rp.t) \not\sqsubseteq t$. Then for some $s \in S$ and $\beta \in \mathbb{E}S$

$$wp.(rp.t).\beta.s > t.\beta.s \ ,$$

whence Definition 7.16

$$\begin{array}{lll}
 & \left(\sqcap\mu : rp.t.s \cdot \int_\mu \beta\right) > t.\beta.s & \\
\implies & \left(\forall\mu : rp.t.s \cdot \int_\mu \beta > t.\beta.s\right) & \\
\text{iff} & \left(\forall\mu : \bar{S} \cdot \left(\forall\beta' : \mathbb{E}S \cdot t.\beta'.s \leq \int_\mu \beta'\right) \implies \int_\mu \beta > t.\beta.s\right) & \text{Definition 7.18} \\
\text{iff} & \left(\cap\beta' : \mathbb{E}S \cdot \left\{\mu : \bar{S} | t.\beta'.s \leq \int_\mu \beta'\right\}\right) \subseteq \left\{\mu : \bar{S} | \int_\mu \beta > t.\beta.s\right\} & \\
\text{iff} & & \text{for arbitrary sets } A, B \text{ write } A \subset B \text{ as } A \cap B^C = \emptyset \\
 & \left(\cap\beta' : \mathbb{E}S \cdot \left\{\mu : \bar{S} | t.\beta'.s \leq \int_\mu \beta'\right\}\right) \cap \left\{\mu : \bar{S} | \beta \leq \int_\mu t.\beta.s\right\} = \emptyset & \\
\text{iff} & & \text{simple algebra, call this line } \dagger \\
 & \left(\cap\beta' : \mathbb{E}S \cdot \left\{\mu : \bar{S} | t.\beta'.s \leq \int_\mu \beta'\right\}\right) & \\
 & \cap \ \left\{\mu : \bar{S} | -t.\beta.s \leq \int_\mu (-\beta)\right\} & \\
 & \cap \ \left\{\mu : \bar{S} | -1 \leq \int_\mu (\underline{-1})\right\} & \\
= & \emptyset &
\end{array}$$

The last line of the above, †, aims to provide a format that can easily be adapted for use with Farkas' lemma (Lemma E.3). However, before this can happen the following is required:

1. the measures and expectations need to be represented in Euclidean space $\mathbb{R}^N$
2. the sets in † need to be shown to be compact
3. a suitable finite set of the measures in † needs to be chosen

Then, together with the finite intersection lemma[6] and Farkas' lemma the proof proceeds as per pGCL [61, Lemma 5.7.4].

The completion of this proof is unresolved because the interaction between these steps cause issues as discussed below.

□

To achieve the first requirement the idea is to use the ϵ-approximation and restrict our attention to measures that are of the format of an ϵ-approximation for some appropriate value of ϵ. This is possible because removing elements from the sets in † will not make the intersection non-empty. The choice of ϵ depends on β and β'. It needs to be small enough to represent all β, β' accurately as a simple function with intervals of width ϵ. Recall (Section 5.1) that Lebesgue integration itself works by using a suitable simple function approximation. Therefore the simple function used to approximate β and β' will use intervals at least as small[7] as those used in the Lebesgue integration of β and β' (so the value of the integrals remain unchanged).

For the third requirement, the appropriate subsets are those determined by restricting β' to the indicator functions on the intervals of ϵ-width that are used in approximating β. Note that to ensure that this set is finite, it is required that all of the measures are only defined over closed intervals of the reals. Together with the first requirement, this converts the problem into an analogous representation to that used for the proof in pGCL.

Finally the finite intersection lemma requires compact sets so it is required to show that all of the sets in † are compact. Unfortunately, it is problematic to show that the sets in the first two lines of † are closed under the Kantorovich metric. This essentially requires that

$$\left\{ \mu \in \bar{S} | a \leq \int_{\mu} g \right\} ,$$

for some real a and function g. Assume that there is some Cauchy sequence (under Kantorovich) in the above, $\mu_0, \mu_1, ...$ To be closed it is required that the limit μ^* is also included, i.e. that

$$\left| \int_{\mu_i} g - \int_{\mu^*} g \right| \to 0 \implies K.\mu_i.\mu^* \to 0 .$$

At first glance this looks fine, as there are obvious similarities between the right hand

[6] The finite intersection lemma states that a collection of closed subsets of a compact set has empty intersection only if some finite sub-collection of it does [72, 61].

[7] At least as small because some of the functions will require smaller intervals than others. The largest possible value of ϵ that accurately represents all functions will be used for all approximations.

side and the Kantorovich metric (Definition 5.11). The issue is, that this only works for functions that satisfy the Lipschitz semi-norm (have a maximum gradient of one everywhere). The indicator functions that are needed for the third requirement do not satisfy this (they have infinite gradient in places).

The completion of this proof is unresolved. Alternative strategies need to be explored, but that is beyond the scope of this thesis. The intention here was just to show the challenge involved in proving consistency for a language containing continuous probability and demonic non-determinism. However, other strategies may include: using a different way of representing measures in Euclidean space; or an alternative to the Kantorovich metric space for comparing measures.

D.2 A Note on the Use of the Kantorovich Metric with Sub-Probability Measures

In this section, an aside on the interaction between the Kantorovich metric and sub-probability (where the total probability assigned may be less than one) is discussed.

There is nothing inherently incompatible between the Kantorovich metric and sub-probability. However, the duality stated in Definition 5.12 (repeated here for convenience):

Definition. Given any two Borel probability measures μ and ν on separable metric space (S, d), the metric L is defined by

$$L.\mu.\nu := \inf\left\{ \int_{S\times S} d.x.y \ \mathrm{d}\gamma.x.y \cdot \gamma \in \Gamma.\mu.\nu \right\} ,$$

where $\Gamma.\mu.\nu$ is the set of all probability measures on $S \times S$ with marginal measures μ and ν. The probability measures μ and ν must satisfy $\int_S d.x.z * \mu.x < \infty$ and $\int_S d.x.z * \nu.x < \infty$ for all $z \in S$ respectively. Note that $\int_{S\times S}$ is used as a shorthand to mean the double integral $\int_S \int_S$. □

breaks down when sub-probability is allowed. This is because it relies on the joint measure $\Gamma.\mu.\nu$. It is not possible for a joint measure to have marginals where the total probability defined for each differs (such probability cannot be created or destroyed in deriving the marginals!).

The definition of the Kantorovich metric for measures over the real numbers (Definition 5.13) also does not hold for measures where the total probability defined differs.

Therefore these two alternative formulations of the Kantorovich metric must be used with care, and avoided where possible. These complications motivated the comment in Lemma D.8, that the weights total of the two measures (μ and ν) had to be the same for their convergence under the Kantorovich metric. That lemma relied on the definition of the Kantorovich metric for real numbers, therefore caution was required to prevent the

comparison of two measures of differing total probability.

The base definition of the Kantorovich metric (Definition 5.11) still holds for sub-probability, however, as does van Breugels version of it (Definition D.11) for one-bounded metric spaces. Therefore, the intention is merely to highlight that care must be taken when using the Kantorovich metric with sub-probability. This may mean that the most intuitive definition of the Kantorovich metric cannot be used for a given problem.

Appendix E

Supplementary Lemmas

This appendix provides some supplementary lemmas from textbooks that are used in proving the consistency of the two semantics of sGCL.

The separating hyperplane lemma is taken from standard linear algebra textbooks [56]. The version presented here is taken from McIver and Morgan's book [61].

Lemma E.1. Let $\mathcal{C}$ be a convex and Cauchy-closed subset of $\mathbb{R}^N$, and p a point in $\mathbb{R}^N$ that does not lie in $\mathcal{C}$. Then there is a separating hyperplane S with p on one side of it and all of $\mathcal{C}$ on the other.

Proof. The reader is referred to linear algebra textbooks [56, p.16] for the proof. □

McIver and Morgan [61] also define a second version of the separating hyperplane lemma for infinite state spaces as follows:

Lemma E.2. Let $\mathcal{C}$ be a convex subset of $\mathbb{R}^S$ that is compact (hence closed) in the product $\mathcal{E}_S$ of the Euclidean topologies over its constituent projections $\mathbb{R}$. If some p does not lie in $\mathbb{C}$, then there is a separating hyperplane with p on one side of it and all of $\mathcal{C}$ on the other.

Proof. The reader is referred to McIver and Morgan's book [61, Lemma B.5.3] for the proof. □

Another important geometric lemma presented by McIver and Morgan [61, Lemma B.5.2] is Farkas' lemma:

Lemma E.3. Let A be and $M \times N$ matrix, x and $N \times 1$ column-vector and r an $M \times 1$ column-vector, and suppose that A and r are so that the system of equations

$$A \cdot x \geq r$$

has no solution in x, where $\cdot$ denotes matrix multiplication. Then there is a $1 \times M$ row-vector C of non-negative values such that

$$C \cdot A = 0 \text{ but } C \cdot r > 0 \ .$$

Proof. The proof can be found in texts on linear programming, for example Schrijver's book [76, Corollary 7.1e]. □

Appendix F

DESTECS Patterns

This appendix provides the two patterns from the DESTECS pattern library [13] to which the monitoring voter pattern described in Chapter 8 is most closely related. Note that some details specific to the DESTECS project have been omitted from these descriptions.

F.1 Voter Pattern

Intent

To produce a single sensor reading from multiple (redundant or diverse) sensor inputs.

Motivation

Where sensors can fail, multiple sensors can be introduced as a way to achieve dependability. This can be done through replication (using copies of the same sensor) or diversity (using different sensors). In order to gain a single value from various inputs, a voter can be used. A simple voter could take the mean of the incoming values, or use a majority vote to ignore erroneous readings.

Structure

A class and object diagram for this pattern are given in Figure F.1. In this pattern, a `Voter` class is introduced that implements the sensor interface (`ISensorReal`). The voter class also aggregates one or more sensors. By implementing the sensor interface, a voter object can be passed transparently to the controller (such that the controller does not need to be altered) and provides a single value from the multiple inputs. The voter pattern can be combined with the *strategy pattern* [29] by describing the voting algorithm as an interface and providing different implementations of this interface to explore alternative voting routines.

The object diagram in Figure F.1 shows that the controller holds a reference to

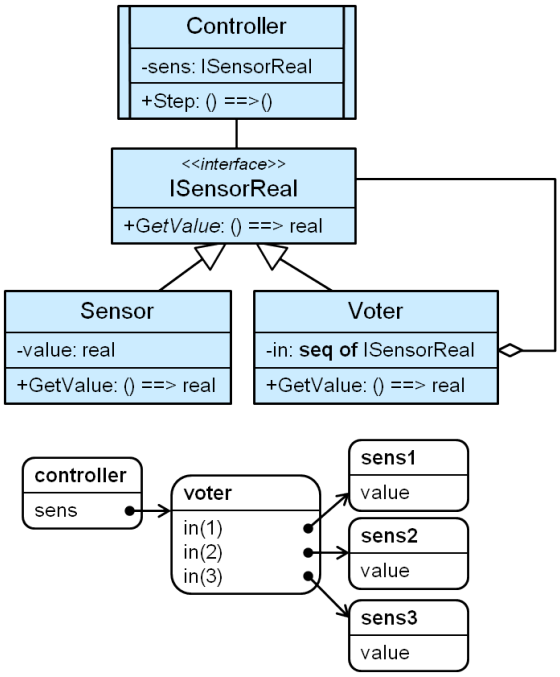

Figure F.1: Voter pattern (class and object diagram)

the voter object, which then aggregates (in this case) three other sensor objects (which represent distinct data sources). When the controller reads its sensor object, the voter decides on the value that the controller receives.

Related Patterns

Filter pattern [13]; strategy pattern [29].

F.2 Monitor

Intent

A monitor (or watchdog) is a small, verifiable component that runs as separate process. It monitors actions of the controller (or other components) and protects from unsafe situations by intervening and instructing the controller to stop the unsafe action.

Motivation

The monitor is modelled as an object that holds a reference to the controller and runs as a separate process on a different CPU. The monitor then checks the actions of the controller and intervenes in some way — for example, by calling an operation on controller that puts it in a safe mode. The monitor could also hold references to other objects to monitor them as required, they typically its scope is limited to keep it small (as with a

kernel).

Structure

A class and object diagram for this pattern are given in Figure F.2. In this pattern, a `Monitor` class is created that holds a reference to the controller. The `Controller` class provides an operation that the monitor can call if some fault occurs (in this case, the controller will go into a safe mode).

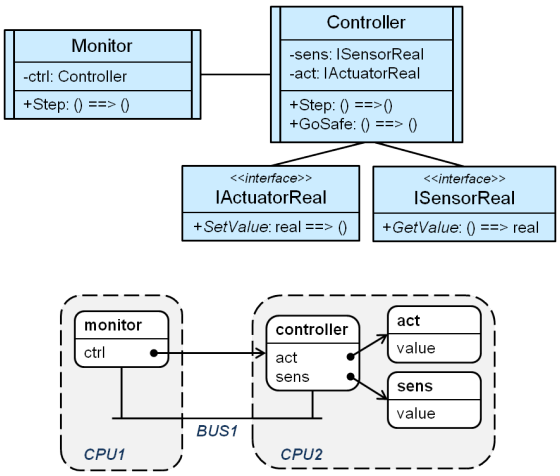

Figure F.2: Monitor pattern (class and object diagram)

The object diagram in Figure F.2 shows that the monitor object holds a reference to the controller object and runs on a separate CPU. It can then put the controller into a safe mode if a fault is detected.

Related Patterns

Kernel pattern [13].

Bibliography

[1] Samson Abramsky and Achim Jung. *Domain theory*, pages 1–168. Oxford University Press, Oxford, UK, 1994.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.

[4] S. Andova. Time and probability in process algebra. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST '00)*, volume 1816 of *LNCS*, pages 323–338, London, UK, UK, 2000. Springer-Verlag.

[5] Z.H. Andrews. Towards a stochastic Event-B. In *Supp. Volume of 2009 Workshop on Quantitative Formal Methods: Theory and Applications*, 2009.

[6] Z.H. Andrews, A. McIver, L. Meinicke, and C. Morgan. Probabilistic aspects of flash filestores. In R. Joshi, T. Margaria, P. Müller, D. Naumann, and H. Yang, editors, *Int. Conf. on Verified Software: Theories, Tools and Experiments 2010, Workshop Proceedings*, 2010.

[7] Robert B. Ash and Catherine Doléans-Dade. *Probability and measure theory*. Academic Press, 2000.

[8] A. Avizienis. The N-version approach to fault-tolerant systems. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, Dec. 1985.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C.E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[10] R.J. Back. Refinement Calculus II: Parallel and reactive programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, pages 67–93, Mook, The Netherlands, May 1989. Springer-Verlag.

[11] R.J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[12] R.P. Boas and H.P. Boas. *A Primer of Real Functions*. The Carus mathematical monographs. Mathematical Association of America, 1996.

[13] Jan F. Broenink, John Fitzgerald, Carl Gamble, Yunyun Ni, Ken Pierce, and Xiaochen Zhang. Methodological guidelines 2. Deliverable D2.2, DESTECS (www.destecs.org), January 2012.

[14] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, 2009.

[15] C.D. Cantrell. *Modern mathematical methods for physicists and engineers*. Cambridge University Press, 2000.

[16] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.

[17] A.C. Coombes, L.M. Barroca, J.S. Fitzgerald, J.A. McDermid, A. Saeed, and L. Spencer. Formal specification of an aerospace system: the attitude monitor. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, chapter 13, pages 307–332. Prentice-Hall International Series in Computer Science, 1995.

[18] R.J. Corin and J.I. den Hartog. A probabilistic Hoare-style logic for game-based cryptographic proofs. In M. Bugliesi, B. Preneel, and V. Sassone, editors, *ICALP 2006 track C, Venice, Italy*, volume 4052 of *Lecture Notes in Computer Science*, pages 252–263, Berlin, July 2006. Springer-Verlag.

[19] K. Damchoom and M. Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In *SBMF 2009*, volume 5902 of *LNCS*, pages 134–152. Springer, 2009.

[20] Yuxin Deng and Wenjie Du. The kantorovich metric in computer science: A brief survey. *Electron. Notes Theor. Comput. Sci.*, 253:73–82, November 2009.

[21] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J. :, 1976.

[22] Ernst-Erich Doberkat. Eilenberg–moore algebras for stochastic relations. *Inf. Comput.*, 204:1756–1781, December 2006.

[23] R. M. Dudley. *Real Analysis and Probability*. Cambridge University Press, 2002.

[24] Richard Durrett. *Probability: Theory and Examples (Probability: Theory & Examples)*. Duxbury Press, 3 edition, mar 2004.

[25] Abbas Edalat. Domain theory and integration. *Theor. Comput. Sci.*, 151(1):163–193, 1995.

[26] J. Falampin. Pilot deployment in transportation. Deliverable (D16), DEPLOY (www.deploy-project.eu), September 2009.

[27] J.S. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-Oriented Systems*. Springer-Verlag, 2005.

[28] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[30] Alison L. Gibbs and Francis Edward Su. On choosing and bounding probability metrics. *International Statistical Review*, 70:419–435, 2002.

[31] Michèle Giry. A categorical approach to probability theory. In *Lect. Notes Maths*, volume 915, pages 68–85. Springer-Verlag, Berlin, 1981.

[32] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[33] S. Hallerstede and T.S. Hoang. Qualitative probabilistic modelling in Event-B. In *Proc. 6th International Conference on Integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 293–312, 2007.

[34] H. Hansson and B. Jonsson. A calculus for communicating systems with time and probabilities. In *Proc. 11th IEEE Real-Time Systems Symp.* IEEE Computer Society Press, December 1990.

[35] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

[36] Osman Hasan, Naeem Abbasi, Behzad Akbarpour, Sofiène Tahar, and Reza Akbarpour. Formal reasoning about expectation properties for continuous random variables. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, pages 435–450, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.

[38] T.S. Hoang. *The Development of a Probabilistic B-Method and a Supporting Toolkit*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, 2005.

[39] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–80, October 1969.

[40] A. Hoefler, J.M. Higman, T. Harp, and P.J. Kuhn. Statistical modeling of the program/erase cycling acceleration of low temperature data retention in floating gate nonvolatile memories. In *Reliability Physics Symposium Proceedings, 2002. 40th Annual*, pages 21–25, 2002.

[41] Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in hol. *Electron. Notes Theor. Comput. Sci.*, 112:95–111, January 2005.

[42] A. Iliasov and A. Romanovsky. Refinement patterns for fault tolerant systems. *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 167–176, May 2008.

[43] He Jifeng, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Sci. Comput. Program.*, 28:171–192, April 1997.

[44] C.B. Jones. Data reification. In J. McDermid, editor, *The Theory and Practice of Refinement*. Butterworths, 1989.

[45] C.B. Jones. *Systematic Software Development using VDM (2nd edition)*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[46] C.B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Ann. Hist. Comput.*, 25(2):26–49, 2003.

[47] C.B. Jones, I.J. Hayes, and M.A. Jackson. Specifying systems that connect to the physical world. Technical Report 964, Newcastle University, School of Computing Science, May 2006.

[48] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Form. Asp. Comput.*, 19(2):269–272, 2007.

[49] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 294–308, Berlin, Heidelberg, 2008. Springer-Verlag.

[50] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[51] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov Reward Model Checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244, Los Alamos, CA, USA, 2005. IEEE Computer Society.

[52] J.-P. Katoen, I.S. Zapreev, E.M. Hahn, H. Hermanns, and D.N. Jansen. The Ins and Outs of The Probabilistic Model Checker MRMC. In *Quantitative Evaluation of Systems (QEST)*, pages 167–176. IEEE Computer Society, 2009.

[53] D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[54] D. Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[55] E. Kreyszig. *Advanced Engineering Mathematics*. Wiley, 8 edition, 1999.

[56] Harold W. Kuhn. *Lectures on the Theory of Games*. Princeton University Press, illustrated edition edition, Jan 2003.

[57] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, April 2002.

[58] J.C. Laprie, J. Arlat, C. Bounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, July 1990.

[59] P.A. Lee and T. Anderson. *Fault Tolerance, Principles and Practice*. Springer-Verlag, 1990.

[60] B. Littlewood, P. Popov, and L. Strigini. Assessing the reliability of diverse fault-tolerant software-based systems. In *SAFECOMP International Conference on Computer Safety, Reliability and Security No19, Rotterdam , PAYS-BAS*, volume 40, pages 781–796, 2002.

[61] A. McIver and C. Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. Springer, 2004.

[62] A. McIver, C. Morgan, and E. Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. In *Proceedings of the International Refinement Workshop and Formal Methods Pacific*, Canberra, 1998.

[63] A. K. McIver and C. Morgan. Partial correctness for probabilistic demonic programs. *Theor. Comput. Sci.*, 266(1-2):513–541, 2001.

[64] L. Meinicke. Personal communication, 2011.

[65] L. Meinicke and I.J. Hayes. Algebraic reasoning for probabilistic action systems and while-loops. *Acta Informatica*, 45(5):321–382, 2008.

[66] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.

[67] C. Morgan. The generalised substitution language extended to probabilistic programs. In *Proc. B'98: The 2nd International B Conference*, volume 1393 of *LNCS*, Montpelier, April 1998.

[68] C. Morgan, T.S. Hoang, and J.-R. Abrial. The challenge of probabilistic Event B - extended abstract. In *Proc. 4th International Conference of B and Z Users (ZB'05)*, pages 162–171, 2005.

[69] C. Morgan, A.K. McIver, K. Seidel, and J. W. Sanders. Refinement-oriented probability for CSP. *Formal Aspects of Computing*, 8(6):617–647, Nov 1996.

[70] K. Pierce, J. Fitzgerald, and C. Gamble. Modelling faults and fault tolerance mechanisms in a paper pinch co-model. In *ERCIM/EWICS/Cyberphysical Systems Workshop at SAFECOMP 2011, Naples*, 2011. Also School of Computing Science, Newcastle University, Technical Report Series 1280.

[71] John A. Rice. *Mathematical Statistics and Data Analysis.* Duxbury Press, 2001.

[72] Halsey Royden. *Real Analysis (3rd Edition).* Prentice Hall, feb 1988.

[73] A.E. Rugina, K. Kanoun, and M. Kaâniche. AADL-based dependability modelling. Technical Report LAAS Research Report 06029, LAAS-CNRS, March 2006.

[74] W.H. Sanders and J.F. Meyer. Stochastic activity networks: Formal definitions and concepts. In *Lectures on Formal Methods and Performance Analysis: first EEF/Euro summer school on trends in computer science*, pages 315–343, New York, NY, USA, 2002. Springer-Verlag New York, Inc.

[75] Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Abstract specification of the UBIFS file system for flash memory. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, pages 190–206, Berlin, Heidelberg, 2009. Springer-Verlag.

[76] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley, June 1998.

[77] I.M. Singer and J.A. Thorpe. *Lecture notes on elementary topology and geometry.* Undergraduate texts in mathematics. Springer-Verlag, 1967.

[78] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. From formal specification in Event-B to probabilistic reliability assessment. In *Proceedings of the 2010 Third International Conference on Dependability*, DEPEND '10, pages 24–31, Washington, DC, USA, 2010. IEEE Computer Society.

[79] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Towards probabilistic modelling in Event-B. In *Proceedings of the 8th international conference on Integrated formal methods*, IFM'10, pages 275–289, Berlin, Heidelberg, 2010. Springer-Verlag.

[80] E. Troubitsyna. *Stepwise Development of Dependable Systems.* PhD thesis, Åbo Akademi University, 2000.

[81] Elena A. Troubitsyna. Reliability assessment through probabilistic refinement. *Nordic J. of Computing*, 6:320–342, September 1999.

[82] Franck van Breugel. The metric monad for probabilistic nondeterminism. Draft available at http://www.cse.yorku.ca/franck/research/drafts/monad.pdf, April 2005.

[83] Cédric Villani. *Optimal Transport: Old and New.* Grundlehren der mathematischen Wissenschaften. Springer, 1 edition, Nov 2008.

[84] N. White. Probabilistic specification and refinement. Master's thesis, Keble College, Oxford, 1996.

[85] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[86] David Woodhouse. JFFS: The Journalling Flash File System. In *Proceedings Ottawa Linux Symposium*, 2001.