School of Computer Science and Engineering
The University of New South Wales

The Development of a Probabilistic B-Method and a Supporting Toolkit

Thai Son Hoang
Bachelor of Engineering, UNSW

A thesis submitted for the degree of
Doctor of Philosophy
July 2005

Supervisor: Associate Professor Ken Robinson
Co-supervisor: Professor Carroll Morgan

Abstract

The *B-Method* (*B*) is a formal method for development of large software systems, and is based on set theory and the predicate calculus. The semantics of *B* is given by the *Generalised Substitution Language* (*GSL*) invented by Abrial, which gives the method a capability of reasoning about the correctness of systems.

Abrial's *GSL* can be modified to operate on arithmetic expressions, rather than Boolean predicates, which allows it to be applied to probabilistic programs. A new operator for probabilistic choice substitution has been added to *GSL* by Morgan, and we get the *probabilistic Generalised Substitution Language* (*pGSL*): a smooth extension of *GSL* that includes random algorithms and probabilistic systems within its scope.

We want to examine the effect of *pGSL* on *B*'s larger-scale structures: its *machines*: for that we suggest a notion of *probabilistic* machine invariants. We show how these invariants interact with *pGSL* at a fine-grained level; and at the other extreme we investigate how they affect our general understanding "in the large" of probabilistic machines and their behaviour.

Furthermore, we want to take these specifications and to refine them into implementations. We present a method that can be used to develop systems with probabilistic properties from specifications to implementations. We give the definition for the consistency of the implementation with respect to the specification, based on the concept of refinement.

Overall, we aim to initiate the development of a *probabilistic B-Method* (*pB*), complete with a suitable *probabilistic Abstract Machine Notation* (*pAMN*). We discuss the practical extension of the *B-Toolkit* to support *pB*, and we give examples to show how *pAMN* can be used to express and reason about probabilistic properties of systems.

Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgment is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Thai Son Hoang
July 2005

Acknowledgements

First and foremost, I would like to thank my supervisor, Associate Professor Ken Robinson for guiding and encouraging me through the ups and downs of the research. I thank him for his patience and suggestions in revising my research work. I am grateful for his support during my trouble times with other non-technical problems. Without his encouragement, I would not have been able to achieve this success.

Secondly, I would like to thank my co-supervisor, Professor Carroll Morgan. I am indebted for his inspiration and motivation, leading the project. I learned a great deal from him in the realistic tackling and solving of problems. And mostly, I thank him for the vision that made me believe in the success of the project.

I would like to thank Dr. Annabelle McIver and Dr. Zhendong Jin, who also participated in this project. You are truly inspirational, the driving forces behind the project. Working with you helped me understand more about the work we shared.

I would like to thank the Australian Research Council for funding the project, the School of Computer Science and Engineering, UNSW, for giving me the scholarship to start with, and the Formal Methods Research Group, National ICT Australia for financially supporting me in the last two years of my Ph.D study. I am fortunate to have been able to participate in such an innovative research group.

I would like to thank B-Core Ltd., especially to Dr. Ib Sørensen, for giving us access to the source code of the *B-Toolkit*.

During my study, I had a chance to work for two months at Royal Holloway, University of London. I am much obligated to Professor Steve Schneider, Dr. Helen Treharne and Dr. Neil Evans for their friendship. We shared the same beliefs and I hope that we will collaborate in the future.

My special thank to Associate Professor Ken Robinson, Professor Carroll Morgan, Dr. Zhendong Jin and Dr. Neil Evans for their time and effort in reading and commenting on this dissertation.

To all Vietnamese friends in Australia, Vietnam and around the World, I would like to thank you for having me as a friend (which sometimes I feel that I let you down). Thank you all for your support and care.

Last but the most, I profoundly thank my parents, my sister and my wife for their constant love, support, encouragement and most of all, having faith in me. I dedicate the work in this dissertation to you all. And the most special thanks to my wife, Thuy Hang, without your support, your caring and most of all, your love, this dissertation would not have been possible.

I would like to thank the examiners for their time and effort. Their comments helped me to improve this dissertation.

Dedication

This thesis is dedicated to my wife who has been a constant source of support, inspiration and motivation.

This thesis is also dedicated to my parents who have been supporting me since the beginning of all my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Mathematics is important not only for computer-hardware engineering but also for software engineering. While mathematics has been more closely connected with the former, there has been significant recent improvements in the application of mathematics to the latter. The application of mathematics is not only of theoretical interest, but has also proved to be successful in practical applications. Formal notations, such as *VDM* [30], *Z* [63] and the *B-Method* (*B*) [4], move software engineering into a new era of a rigorous approach to program developments, especially to critical software where failure is not an option.

B contains a systematic method for development of large software systems from reusable fragments, an essential requirement of software engineering. Based on the *Generalised Substitution Language* (*GSL*) [4] invented by Abrial, *B* offers the capability of specifying and reasoning about software systems using set theory and the predicate calculus.

The *B-Method* is built around the concept of having abstract machines. A machine contains the state of a construct (either specification, refinement or implementation), and also provides operations that change the state and possibly return one or more results. The state of a machine is constrained by its invariant. The consistency of a machine is concerned with the establishment of the invariant by the initialisation and the maintenance of the invariant by the machine's operations. The syntax of *B* machines is given in the *Abstract Machine Notation* (*AMN*), and

the meaning of *AMN* operations is given by *GSL* semantics. The *B-Method* also provides full reasoning about the process of refinement from specifications to implementations. This guarantees that the code generated from the implementation is consistent with the original specification.

Morgan [45] extends Abrial's *GSL* with a new probabilistic choice operator to create *probabilistic Generalised Substitution Language* (*pGSL*), a simple extension which can handle probability. Using *pGSL*, one can specify the behaviour of systems with probabilistic properties.

Essentially, we need to have a method to reason formally about probabilistic systems. We want to develop a *probabilistic B-Method* (*pB*) which includes developing a new syntax and semantics of a *probabilistic Abstract Machine Notation* (*pAMN*); an extension of *AMN* accommodating probability. Then *pB* should be able to operate in the same framework as *B*, i.e. having machine encapsulation and a concept of refinement. Applications of *pB* could include distributed algorithms, in which probability is used to break symmetry, or a system of sensors that can have a certain probability of failure. With the introduction of *pB*, we can formally specify, implement and also reason about the correctness of such systems.

Further, the success of formal methods depends largely on tool support. Without tools, formal methods cannot be applied successfully. For *B*, a number of tools have been built in order to incorporate its ideas; we will use the *B-Toolkit* [37] as our supporting tool. We will modify the *B-Toolkit* in order to support the extension from *B* to *pB*.

This dissertation is concerned with the development of the *pB* method and its associated supporting tool.

1.2 Aims and Contributions

The aim of this dissertation is to investigate the new syntax and semantics for *pB*, which is the extension of *B* based on *pGSL*. The concept of an abstract machine has to be extended to incorporate probability, with new constructs of *pAMN*. The meaning of each construct needs to be studied and developed. Moreover, the question of how probabilistic properties of systems can be expressed in the new framework needs to be answered. The rules for reasoning about the correctness of systems based on the new constructs have to be developed and successfully applied

to various systems with probabilistic properties. The process of refinement from specifications to implementations has to be maintained since this is the backbone of the *B-Method*. More importantly, practical issues need to be taken into account when developing the new method. Whenever possible, we want to retain the original predicate reasoning, while making the necessary extension required for the new elements. The *B-Toolkit* needs to be changed accordingly, and these changes need to be illustrated with examples. The specific contributions of this dissertation are:

- Supporting systems with *almost-certain* termination. In some systems, termination is not guaranteed (certainly), but the probability of these systems terminating is indeed 1. Reasoning about these systems requires a special treatment of probability. The application of this kind of work includes distributed systems, in which probability is used to break the symmetry of system processes efficiently [41].

- Proposing the idea of a probabilistic invariant, which supports the idea of having "probabilistic abstract machines". The informal meaning of a such invariant, i.e. what it guarantees, and the proof obligations for its maintenance, are given [23].

- Introducing probabilistic specification substitutions and the fundamental theorem for refining those substitutions. This work helps to specify and produce consistent code using the concept of refinement. We also discuss the practical issues when extending the *B-Toolkit* to support probabilistic specification substitutions [25].

- Extending probabilistic specification substitutions to systems with multiple probabilistic properties. This gives a more complete coverage of specifications, including some systems that cannot be captured by single probabilistic specification substitutions on their own. Also, the fundamental theorem is widened to cover the refinement of multiple probabilistic specification substitutions.

All the work above is demonstrated with supporting case studies. The publications produced with respect to the research work described in this dissertation can be found in Appendix E.

Development	Version of Toolkit	Related chapter
Root-Contention	The *qB-Toolkit*	Chap. 3
Rabin's Choice-Coordination Algorithm	The *qB-Toolkit*	Chap. 3
Probabilistic Library	The *pB-Toolkit*	Chap. 4
Min-Cut algorithm	The *pB-Toolkit*	Chap. 5
Probabilistic primary testing	The *pB-Toolkit*	Chap. 5
Two Duelling Cowboys	The *pB-Toolkit*	Chap. 6
Three Duelling Cowboys	The *pB-Toolkit*	Chap. 6

Table 1.1: List of complete developments

In Tab. 1.1 are the list of developments that have been developed and proved using the modified toolkits.

1.3 Dissertation Organisation

This dissertation addresses the issues in the development of *pB* and the *pB-Toolkit*, the supporting toolkit. New constructs are introduced into *pAMN* and the *pB-Toolkit* is extended to support their syntax and semantics. *pB* will allow developers to reason about the correctness of probabilistic systems developed using the new toolkit. A brief overview of the background and related work to this dissertation are given in Chap. 2. Where necessary, issues relating to a specific area are given in separate chapters.

In Chap. 2, we review the underlying theoretical work to support the development of *pB*. We review the syntax and semantics of *GSL* and the original *B-Method*, and also discuss the supporting *B-Toolkit*. Furthermore, we give an introduction to *pGSL*, which is the basis of this dissertation.

In Chap. 3, we consider a small extension of the *B-Method* to support a set of systems with probability-one, that is almost-certain termination. We explain the concept of probability-one termination and its theoretical basis. We discuss the extension to the *B-Toolkit* and, finally, develop some examples to show the application of the new method.

In Chap. 4, we introduce the idea of having probabilistic invariant for probabilistic machines. The informal meaning of the new construct is given using a simple example. We also investigate some subtle issues that make a clear distinction between the probabilistic and non-probabilistic domain.

In Chap. 5, we take probabilistic developments into a new level by introducing probabilistic specification substitutions and the fundamental theorem for the refinement of such specifications. We concentrate on practical issues when extending the *pB-Toolkit* to support a well known example in the new framework. We show that development via layers is still valid with the new substitution.

In Chap. 6, we extend probabilistic specification substitutions further to cover systems with multiple expectations. An example is used to illustrate the new substitution and the corresponding fundamental theorem which supports the development of implementations from specifications.

In Chap. 7, we give a summary of the work and discuss possible further research relating to this dissertation.

Chapter 2

Background and Related Work

This chapter introduces the components of the *B-Method* (*B*): the *Generalised Substitution Language* (*GSL*) that provides a formal semantics and the *Abstract Machine Notation* (*AMN*) that provides the concept of an "abstract machine" — a basic construct within the method. This provides an understanding of the *B-Method* and the supporting *B-Toolkit*. We also introduce the *probabilistic Generalised Substitution Language* (*pGSL*), which forms the basis of the research described in this dissertation.

2.1 The Generalised Substitution Language

We first briefly review *GSL*; then we look at its syntax and semantics, together with some examples.

2.1.1 Meaning of Programs

Programming languages usually have a formal syntax and an informal semantics. This leads to difficulty or imprecision in verifying the correctness of programs written in these languages. Giving programs a precise meaning requires a formal semantics of the language constructs.

Starting with Hoare triples in the late 1960's [26], the meaning of a program is defined in terms of a precondition and post-condition pair as follows:

$$\{P\} \quad S \quad \{Q\} \,.$$

In the above formula, P and Q are predicates of the state of program S. The meaning of the triple is that the program S starts from initial state satisfying P and guarantees to establish Q in the final state.

Later, Dijkstra shifted the focus to total correctness and made the process more "goal directed" [16]. He introduced the definition of a "weakest precondition", denoted by $wp(S, Q)$, where S is a program and Q is a predicate of the program state. The meaning of the weakest precondition is that it characterises the state under which the program S guarantees to establish the post-condition Q. An equivalent form of the Hoare triples

$$\{P\} \quad S \quad \{Q\}$$

would be expressed in *wp* notation as

$$P \quad \Rightarrow \quad wp(S, Q) \; .$$

Based on weakest preconditions, Dijkstra defined the semantics of the *Guarded Command Language* (*GCL*) which assigns a meaning to different commands including assignment, guarded commands, precondition commands, conditional commands and loops [16].

Abrial's *GSL* [4] is another notation for assigning meaning to programs, and is close to Dijkstra's *GCL*. In *GSL*, a program is understood as a "predicate transformer", which relates the before and after states. Consider the simple program:

$$x := \; y - 1 \; , \tag{2.1}$$

that assigns the value $y - 1$ to x. We want to know under which precondition the state, after executing Prog. (2.1), satisfies the post-condition $x \in \mathbb{N}$. It turns out that the answer is just this predicate in which we "substitute" [1] $y - 1$ for x, i.e. $y - 1 \in \mathbb{N}$. We begin with the definition of *GSL* for simple substitutions using the notation $[]$, which is equivalent to $wp(,)$ notation, as follows:

$$[x := \; E] \, P \quad \widehat{=} \quad \text{"}P \text{ with every free occurrence of } x \text{ replaced by } E\text{"} \; .$$

Moreover, *GSL* allows elements of the language to be composed, i.e. the meaning of a substitution is defined by the meaning of its individual smaller components

[1] This is the reason why the meanings of programs in *GSL* are defined in terms of *substitutions*.

and the way that they composed. In order to be suitable as a specification language, *GSL* provides non-deterministic substitutions and leaves the implementation issues to later stages of the development. Consider the following program:

$$x := y - 1 \quad [\!] \quad x := z - 1 \,, \tag{2.2}$$

which assigns x to either $y - 1$ or $z - 1$. In this case, the choice of which branch to run is up to the implementor of the system to decide. This non-deterministic substitution shows the capability of abstraction within the language. Users of the program should be happy with either one of the branches, and that leaves the implementor with a choice of which branch to implement. It means that no matter which branch is chosen, the corresponding substitution must establish the desired post-condition. Hence, the semantics for the non-deterministic choice substitution is as follows:

$$[S \ [\!] \ T]\, P \quad \widehat{=} \quad [S]\, P \ \wedge \ [T]\, P \,.$$

For example, again, we consider the post-condition $x \in \mathbb{N}$, i.e. we want to know under which precondition the state, after executing Prog. (2.2), satisfies this condition. Since both branches of the program need to satisfy this post-condition, we have that the precondition is

$$[x := y - 1 \quad [\!] \quad x := z - 1]\,(x \in \mathbb{N})$$

$\equiv$ non-deterministic substitution

$$[x := y - 1]\,(x \in \mathbb{N}) \quad \wedge \quad [x := z - 1]\,(x \in \mathbb{N})$$

$\equiv$ simple substitution

$$y - 1 \in \mathbb{N} \quad \wedge \quad z - 1 \in \mathbb{N}$$

This result is consistent with what we expect from the meaning of the above substitution. Since the user must be happy with either of the branches, we must make sure that the substitution is correct for both of them.

2.1.2 *GSL* Syntax

A summary of *GSL* syntax is given in Fig. 2.1 on the following page. As in other common programming languages, *GSL* has assignment (simple substitution) and

$x := E$	x is assigned value E.
$P \mid S$	With precondition P, substitution S is executed.
$S \ [\!] \ T$	Non-deterministic choice between substitutions S and T.
$S \mid\mid T$	Parallel substitution: S and T are executed concurrently.
$S; T$	Sequential substitution: S is executed first, and then T.
$P \Longrightarrow S$	Substitution S is executed only if the guard P holds.
skip	skip substitution (do nothing).
$@\, x \cdot S$	Unbounded-choice substitution.

- P is a predicate over state variables;
- S, T are generalised substitutions;
- x is a variable (or a vector of variables);
- E is an expression (or a vector of expressions).

Figure 2.1: *GSL* syntax

sequential substitution. But we also have substitutions with preconditions, non-deterministic choice, unbounded choice and parallel substitutions within *GSL*.

As an example for a program written in *GSL*, we have the following program:

$$(x \in \mathbb{N} \ \wedge \ x \neq 0) \quad \mid \quad x := x - 1 \,, \tag{2.3}$$

that decreases x by 1 under the precondition that x is a non-zero natural number. The precondition is just the assumption under which the substitution is carried out. We can also have guarded substitutions as in the following example:

$$(x \in \mathbb{N} \ \wedge \ x \neq 0) \quad \Longrightarrow \quad x := x - 1 \,, \tag{2.4}$$

which decreases x by 1 only if x is a non-zero natural number.

There is a clear difference between a preconditioned substitution and a guarded substitution. Consider the following substitutions

$$P \mid S \tag{2.5}$$

and

$$G \Longrightarrow S\,. \tag{2.6}$$

In (2.6), the substitution S is executed only if the condition G is satisfied. On the other hand, in (2.5), the substitution S is always carried out; but in the case where P does not hold, the preconditioned substitution does not guarantee anything on the outcome.

2.1.3 *GSL* Semantics

A summary of the semantics of *GSL* substitutions is given in Fig. 2.2 on the next page. The semantics of *GSL* is expressed in terms of the weakest precondition with respect to an arbitrary predicate Q.

As an example, we consider Prog. (2.3) with the post-condition that $x \in \mathbb{N}$:

$$[(x \in \mathbb{N} \wedge x \neq 0) \mid x := x - 1]\ (x \in \mathbb{N})$$

$$\equiv \quad x \in \mathbb{N} \wedge x \neq 0 \wedge [x := x - 1]\,(x \in \mathbb{N}) \qquad \text{preconditioned substitution}$$

$$\equiv \quad x \in \mathbb{N} \wedge x \neq 0 \wedge x - 1 \in \mathbb{N} \qquad \text{simple substitution}$$

$$\equiv \quad x \in \mathbb{N} \wedge x \neq 0\,. \qquad \text{logic}$$

And if we consider Prog. (2.4) with the same post condition, we have:

$$[(x \in \mathbb{N} \wedge x \neq 0) \Longrightarrow x := x - 1]\ (x \in \mathbb{N})$$

$$\equiv \quad x \in \mathbb{N} \wedge x \neq 0 \Rightarrow [x := x - 1]\,(x \in \mathbb{N}) \qquad \text{guarded substitution}$$

$$\equiv \quad x \in \mathbb{N} \wedge x \neq 0 \Rightarrow x - 1 \in \mathbb{N}\,. \qquad \text{simple substitution}$$

$[x := E]\, Q$	The predicate obtained by replacing all free occurrences of x by E in Q.
$[P \mid S]\, Q$	$P \ \wedge\ [S]\, Q$.
$[S \ [\!]\ T]\, Q$	$[S]\, Q \ \wedge\ [T]\, Q$.
$[S; T]\, Q$	$[S]\, ([T]\, Q)$.
$[P \Longrightarrow S] Q$	$P \Rightarrow ([S]\, Q)$.
$[\mathsf{skip}]\, Q$	Q .
$[@\, x \cdot S]\, Q$	$\forall\, x \cdot [S]\, Q$.

- P, Q are predicates;
- S, T are generalised substitutions;
- x is a variable (or a vector of variables).
- E is an expression (or a vector of expressions).

Figure 2.2: *GSL* semantics

$$\equiv \quad \mathsf{true}\ . \qquad \text{logic}$$

A more complex example, where the program comprises guarded substitutions and a non-deterministic choice, is as follows:

$$\begin{array}{rl} & y \geq 1 \implies x := y - 1 \\ [\!] & y < 1 \implies x := y\ . \end{array} \tag{2.7}$$

This is in fact a conditional substitution: if $y \geq 1$, x is assigned the value $y - 1$, else (if $y < 1$), x is assigned the value y. Again, considering the post-condition $x \in \mathbb{N}$, we have:

$$\left[\begin{array}{rl} & y \geq 1 \implies x := y - 1 \\ [\!] & y < 1 \implies x := y \end{array}\right] (x \in \mathbb{N})$$

$$\begin{array}{rll} \equiv & [y \geq 1 \implies x := y - 1]\ (x \in \mathbb{N}) & \text{non-deterministic choice} \\ \wedge & [y < 1 \implies x := y]\ (x \in \mathbb{N}) & \end{array}$$

$$\begin{array}{lll} \equiv & y \geq 1 \Rightarrow [x := y - 1]\ (x \in \mathbb{N}) & \text{guarded substitution} \\ \wedge & y < 1 \Rightarrow [x := y]\ (x \in \mathbb{N}) & \end{array}$$

$$\begin{array}{lll} \equiv & y \geq 1 \Rightarrow y - 1 \in \mathbb{N} & \text{simple substitution} \\ \wedge & y < 1 \Rightarrow y \in \mathbb{N}\,. & \end{array}$$

The meaning of Prog. (2.7) is shown by decomposing it into simpler substitutions. This is common in *GSL* to get the meaning of a complex substitution according to its components and the way they are composed.

2.2 The Abstract Machine Notation

The *B* notation is based on the the concept of *abstract machines* which keep the states of programs and provides operations for changing the states. The meaning of abstract machines' operations is given by *GSL* semantics, which is introduced in the previous section.

We start by looking at a simple abstract machine, and add more details to it. A simple abstract machine can be seen in Fig. 2.3 on the following page. The MACHINE clause introduces the name of the specification (machine). A specification can also have parameters declared with the MACHINE clause. The VARIABLES clause introduces the state of the machine, which is a set of variables. The INVARIANT clause sets some constraints on the variables, which must be preserved by every operation in the machine (and established by the initialisation). The INITIALISATION clause is the initial setup for the machine, i.e. giving the starting values for all variables. The last clause in the example, the OPERATIONS clause, contains a list of various operations, which are used to change the state of the machine.

In a machine, the body of the operation can be specified using substitutions. For example, the Increase operation in Fig. 2.3 on the next page could be specified as follows:

$$num := num + 1\,.$$

The operation needs to maintain the invariant of the machine, under the assumption that the invariant holds, i.e.

$$num \in \mathbb{N} \quad \Rightarrow \quad [num := num + 1]\,(num \in \mathbb{N})\,,$$

MACHINE *Number*
VARIABLES *num*
INVARIANT $num \in \mathbb{N}$
INITIALISATION $num := 0$
OPERATIONS
  Increase $\widehat{=} \cdots$ **;**
  Decrease $\widehat{=} \cdots$
END

Figure 2.3: A simple machine

or equivalently

$$num \in \mathbb{N} \quad \Rightarrow \quad num + 1 \in \mathbb{N} \,,$$

which is obviously true.

For the Decrease operation, we intend to decrease the value of num by 1, i.e. by having the substitution:

$$num := \ num - 1 \,.$$

Of course, this substitution does not guarantee to preserve the invariant of the machine. More precisely, when $num = 0$, the operation Decrease will break the invariant. The substitution needs to have a precondition (assumption), under which the operation can be guaranteed to be consistent. With this informal reasoning, we see that the precondition that we need to add is:

$$num \neq 0 \,.$$

In *AMN*, we would therefore write the specification of the Decrease operation as follows:

Decrease $\widehat{=}$
  PRE $num \neq 0$
  THEN $num := num - 1$
  END

We now need to adjust the proof obligation slightly, because the operation is assumed to be executed under a non-trivial precondition. We say that the operation

guarantees to re-establish the invariant of the machine under the assumption that the invariant *and the precondition* of the operation both hold. For the Decrease operation, we have to prove that:

$$num \in \mathbb{N} \ \wedge \ num \neq 0 \quad \Rightarrow \quad [num := \ num - 1]\,(num \in \mathbb{N})\,,$$

or equivalently,

$$num \in \mathbb{N} \ \wedge \ num \neq 0 \quad \Rightarrow \quad num - 1 \in \mathbb{N}\,,$$

which is obviously true.

Table 2.1: Abstract Machine syntax

Term	Definition
Machine	**MACHINE** *MachineDeclaration* **CONSTRAINTS** *Predicate* **USES** *MachineList* **SEES** *MachineList* **INCLUDES** *InstantiatedMachineList* **PROMOTES** *OperationNameList* **EXTENDS** *InstantiatedMachineList* **SETS** *SetList* **CONSTANTS** *ConstantList* **PROPERTIES** *Predicate* **VARIABLES** *VariableList* **INVARIANT** *Predicate* **ASSERTIONS** *Predicate* **DEFINITIONS** *DefinitionList* **INITIALISATION** *Substitution* **OPERATIONS** *OperationList* **END**
MachineDeclaration	*Machine* *Machine*(*ParameterList*)
MachineList	*Machine* *MachineList*, *Machine*
InstantiatedMachineList	*InstantiatedMachine* *InstantiatedMachineList*, *InstantiatedMachine*
InstantiatedMachine	*Machine* *Machine*(*ActualParameterList*)
ActualParameterList	*ActualParameter* *ActualParameterList*, *ActualParameter*

Continued on next page

Table 2.1 — continued from previous page

Term	Definition
$SetList$	Set $SetList; Set$
Set	$SetName$ $SetName = \{EnumerationList\}$
$DefinitionList$	$Definition \mathrel{\widehat{=}} Expr$ $DefinitionList; Definition \mathrel{\widehat{=}} Expr$

The abstract syntax of *abstract machines* is defined in Tab. 2.1. Notice that, with the exception of the MACHINE clause at the beginning and the END clause at the end, all other clauses are optional.

An "abstract machine" starts with the MACHINE clause which gives the name and possible parameters of the specification. The CONSTRAINTS clause states the assumption of the parameters.

Machines can be composed in order to produce larger systems. In *B*, there are four different ways to compose machines: SEES, USES, INCLUDES and EXTENDS clauses.

- SEES clause: gives read-only access to other machines. This is restricted to the sets, constants and definitions of the seen machine.
- USES clause: gives read-only access to other machines. This allows the using machine to refer to the variables of the used machine in the invariant and precondition of its operations. The condition for this clause is that both used and using machines need to be "included" in another machine.
- INCLUDES clause: the included machine becomes part of the including machine. The including machine can modify the state of the included machine, but only by calling the operations of the included machine. The PROMOTES clause indicates the list of operations in the included machine that are "promoted" to become operations of the including machine.
- EXTENDS clause: a machine "extends" another machine by *including* the machine and *promoting* all operations of the extended machine.

When *including* (*extending*) another machine, the parameters of the included (extended) machine must be instantiated.

To declare sets in *B*, there is the SETS clause. There are two types of sets in *B*: deferred sets and enumerated sets. The former are declared by giving only their names in the SETS clause, whereas the latter are declared by also giving the elements of those sets.

Constants are declared using the CONSTANTS clause and the properties of constants are given using the PROPERTIES clause. The PROPERTIES clause can also give constraints on the sets.

The state of the machine is kept by a set of variables which are declared in the VARIABLES clause. The INVARIANT clause imposes constraint on the state of the machine, and it must be true at anytime during the execution of the system, i.e. after the initialisation and every operation. The proof obligations of operations are concerned with maintaining such an invariant. The ASSERTIONS clause also constrains the variables of the machine, but it can be derived from the INVARIANT. The purpose of having the ASSERTIONS clause is only to help with proving the obligations.

The INITIALISATION clause is a substitution that gives initial values for variables of the system. These values also need to satisfy the invariant of the system.

The OPERATIONS clause lists all the operations of the system that can be invoked in order to change the state. The syntax of an operation in *B* is as follows:

$$
\begin{array}{l}
outputs \longleftarrow Op(inputs) \ \widehat{=} \\
\textbf{PRE}\ Predicate\ \textbf{THEN} \\
\quad Substitution \\
\textbf{END}
\end{array}
$$

An operation can have some inputs and/or some outputs. The *precondition* of an operation not only constrains the state of the machine, but also gives the condition on the inputs of the operation. The *substitution* is defined to change the state of the machine and provide output. In *B*'s operations, the *inputs*, *outputs* and *precondition* are optional. Further details of the syntax of *B* machines can be found in [4, 68, 60, 33, 32].

A summary of *AMN* substitutions, together with their definitions in *GSL*, is shown in Tab. 2.2. In the last construct in the table (called choice from a set), we need to have a side condition $y \setminus E$ that says y is not free in E.

Table 2.2: From *AMN* substitutions to *GSL*

AMN substitution	*GSL* equivalent
BEGIN S *END*	S
PRE P *THEN* S *END*	$P \mid S$
$x := E \;\|\|\; y := F$	$x, y := E, F$
CHOICE S *OR* T *END*	$S \;[]\; T$
IF P *THEN* S *ELSE* T *END*	$(P \implies S) \;[]\; (\neg P \implies T)$
IF P *THEN* S *END*	$(P \implies S) \;[]\; (\neg P \implies \mathsf{skip})$
ANY x *WHERE* P *THEN* S *END*	$@\, x \cdot (P \implies S)$
$x :\in E$	$@\, y \cdot (y \in E \implies x := y)$

More syntactical extensions of *AMN* can be defined accordingly, such as *ELSIF*, *CASE* and *SELECT* clauses. These extensions can be found in [4, 68].

Notice that the parallel substitution does not have any precise semantics; instead, it is syntactically de-sugared and ultimately becomes a multiple substitution. For example, a parallel substitution that involves two simple substitutions

$$x := 1 \;\|\; y := 1\,, \tag{2.8}$$

is de-sugared to

$$x, y := 1, 1\,. \tag{2.9}$$

Some of the rules for de-sugaring parallel substitutions are shown in Tab. 2.3.

Table 2.3: De-sugaring parallel substitution

Left	Right (de-sugared
$(S \ [\!] \ T) \ \|\| \ U$	$(S \ \|\| \ U) \ [\!] \ (T \ \|\| \ U)$
$U \ \|\| \ (S \ [\!] \ T)$	$(U \ \|\| \ S) \ [\!] \ (U \ \|\| \ T)$
$S \ \|\| \ (T \ \|\| \ U)$	$S \ \|\| \ T \ \|\| \ U$
$S \ \|\| \ T \ \|\| \ \mathsf{skip}$	$S \ \|\| \ T$
$S \ \|\| \ \mathsf{skip} \ \|\| \ U$	$S \ \|\| \ U$
$\mathsf{skip} \ \|\| \ T \ \|\| \ U$	$T \ \|\| \ U$

For a complete formal definition of *AMN*, the reader is referred to [4].

2.3 Software Development Using the *B-Method*

The *B-Method* is a method for formal developments of large software systems, which is an essential technique in software engineering. By using the idea of abstract machines for describing the explicit model of the states of systems, the abstraction captures the essential properties of the data, but not necessarily the implementation details. Formal developments enable the developers with a power that is not available to those who are using informal methods: the power to prove the correctness of systems. As described in Sec. 2.2, the abstract machines are entirely based on set theory and predicate calculus, and proof obligations are generated accordingly to guarantee their consistencies, which will be crucial for some software systems, such as defence or safety-critical systems.

In *B*, the abstract machine has three forms: *specification*, *refinement* and *implementation*, that are used at different stages of the development process. This process can be seen in Fig. 2.4 on the following page.

The development process starts with a specification, in which the operations can be written using abstract constructs, including parallel substitutions and non-deterministic substitutions. The invariant of the specification gives the relationships between those variables that make up the state of the system. Together with the initialisation and operations, they are part of the state-machine model of software systems.

In the next step, the developer needs to “refine” the specification because of its

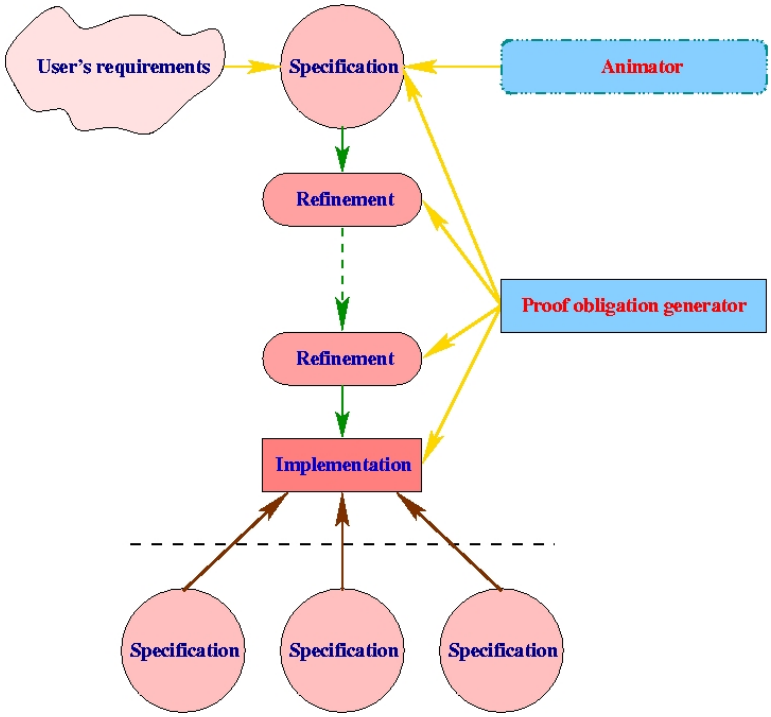

Figure 2.4: Software development using the *B-Method*

abstraction. There are no executable code equivalents to parallel or non-deterministic substitutions. This step can be repeated so that the structures of data and operations become more concrete and closer to executable code. More details can be added to the development and non-translatable constructs can be resolved during this sequence of *refinements*.

At the end of this process, an *implementation* is created. It is a special refinement, in which only limited constructs can be used. This is because we want to be able to translate from the implementation into executable code directly. It means that constructs such as parallel substitutions and non-deterministic substitutions cannot be present in the implementation.

During the development, proof obligations are generated and discharged in order to maintain the correctness of the refinement. The obligations will provide the consistency of transforming from a simple abstract specification to a complete implementation of a system. This is the main advantage of using formal methods and in particular of using *B* where the correctness for a whole development can be assured.

MACHINE *Set*
SEES *Bool_TYPE*
VARIABLES *set*
INVARIANT $set \subseteq \mathbb{N}$
INITIALISATION $set := \{\}$

OPERATIONS

Operation: AddSet
Requirements: Add an element *elem* to the set.
Precondition: The element is new.

AddSet (*elem*) $\widehat{=}$
PRE $elem \in \mathbb{N} \wedge elem \notin set$ **THEN**
$set := set \cup \{ elem \}$
END ;

Operation: InSet
Requirements: Check whether an element *elem* is in the set or not and return *TRUE* or *FALSE* accordingly.
Precondition: Trivial precondition about type of the input.

$in \longleftarrow$ **InSet** (*elem*) $\widehat{=}$
PRE $elem \in \mathbb{N}$ **THEN**
IF $elem \in set$ **THEN** $in := TRUE$
ELSE $in := FALSE$
END
END
END

Figure 2.5: Specification of a simple set

As a first example of a development using the *B-Method*, we consider a specification of a simple set of natural numbers, with two operations. The first operation is to add an element to the set. The second one is a query operation to check whether an element is in the set or not. The specification is shown in Fig. 2.5.

REFINEMENT *SetR*
REFINES *Set*
SEES *Bool_TYPE*
VARIABLES *sequence*
INVARIANT
$sequence \in \mathsf{iseq}(\mathbb{N}) \wedge$
Retrieval relation
$\mathsf{ran}(sequence) = set$
INITIALISATION
$sequence := [\,]$

OPERATIONS

Operation: AddSet
Requirements: Append the element to the end of the sequence.

AddSet (*elem*) $\widehat{=}$ $sequence := sequence \leftarrow elem$ **;**

Operation: InSet
Requirements: Check whether the element is in the range of the sequence or not and return *TRUE* or *FALSE* accordingly.

$in \longleftarrow$ **InSet** (*elem*) $\widehat{=}$
IF $elem \in \mathsf{ran}(sequence)$ **THEN** $in := TRUE$
ELSE $in := FALSE$
END
END

Figure 2.6: Refinement of a simple set by a sequence

We refine the set to an injective sequence which contains all the elements of the set. A refinement construct starts with the REFINEMENT clause which gives the name of the construct (with no parameter), and is followed by the REFINES clause which states the name of the construct that it refines. The rest of a refinement construct's syntax is the same as a machine's syntax. The invariant now also

states a retrieval relationship which relates the variables of the refined and refining machines. The proof obligations of refinement constructs are not only concerned with maintaining the invariant but also the refinement relationship between the operations of the refining and refined constructs. Because of changes in data representation, the operations are now modelled in terms of the sequence. The syntax of the refinement's operations is nearly the same as syntax specification's operations (as described in Sec. 2.2), with a small extension because sequential composition is now allowed. The refinement is shown in Fig. 2.6 on the preceding page.

An implementation is a special refinement. In *B*, an implementation starts with the IMPLEMENTATION clause which gives the name of the construct and the REFINES clause which gives the name of the construct that it implements. The implementation does not have a state of its own. Instead, it "imports" other specifications to model the state of the refined construct. In our case, a "modified" [2] copy of the system library for modelling a sequence of natural numbers (namely nat_Nseq) is used for implementing the system. The operations are now implemented by calling operations from the *imported* machine accordingly. The implementation is shown in Fig. 2.7 on the following page. A part of the supporting machine nat_Nseq related to the implementation can be seen in Fig. 2.8 on page 25.

Certainly, a specification can be refined and implemented in many levels and developers can reuse specifications of their own for the purpose of implementing other specifications.

The software development through refinement and implementation guarantees that the final executable code is proved to be consistent with the specification.

2.4 The *B-Toolkit*

The *B-Method* is supported by the *B-Toolkit* [37]. Other toolkits are also supporting the *B-Method* are Atelier B [14], B4free [15], jBTools [12] and StudioB [13]. We have chosen to use the *B-Toolkit* as our main toolkit to illustrate the extension of the method to a probabilistic context.

The *B-Toolkit* is developed by B-Core (UK) Ltd. [37]. It is responsible for

[2] The actual system library does not support unbounded natural numbers and unbounded sequences. In our case, in order to use the actual system library, the development would have to introduce bounds from the specification level.

IMPLEMENTATION *SetRI*
REFINES *SetR*
SEES *Bool_TYPE*
IMPORTS *nat_Nseq*
INVARIANT

Retrieval relation

$nat\_Nseq = sequence$

OPERATIONS

Operation: AddSet
Requirements: Implementation of the operation by calling the PSH_NSEQ operation of the imported machine.

AddSet (*elem*) $\widehat{=}$ *nat_PSH_NSEQ* (*elem*) **;**

Operation: InSet
Requirements: Using the SCH_LO_EQL_NSEQ operation of the imported machine to find if the element is in the sequence or not.

in $\longleftarrow$ **InSet** (*elem*) $\widehat{=}$
VAR *jj* , *kk* , *ii* **IN**
jj := *1* **;**
kk $\longleftarrow$ *nat_LEN_NSEQ* **;**
in , *ii* $\longleftarrow$ *nat_SCH_LO_EQL_NSEQ* (*jj* , *kk* , *elem*)
END
END

Figure 2.7: Implementation using a system library

maintaining the configuration of a development, in which programs are put through different phases: *analysis*, *proof-obligation generation*, *proof of obligations*, *documentation*, etc. The internals of the Toolkit are built on top of a theorem prover called the "B-Platform" (or sometimes, the "B-Tool"). The Toolkit is controlled by a set of "Toolkit binaries" which is the set of rules to drive all of the *B-Toolkit*'s

MACHINE *nat_Nseq*
...
VARIABLES *nat_Nseq*
INVARIANT
nat_Nseq ∈ seq (ℕ)
INITIALISATION
nat_Nseq := [ ]

OPERATIONS
...
nat_PSH_NSEQ (*vv*) ≙
PRE
vv ∈ ℕ
THEN
nat_Nseq := *nat_Nseq* ← *vv*
END ;
...
bb , *ii* ⟵ **nat_SCH_LO_EQL_NSEQ** (*jj* , *kk* , *vv*) ≙
PRE
vv ∈ ℕ ∧
jj ∈ ℕ ∧
kk ∈ ℕ
THEN
LET *ss* **BE**
ss = *jj* . . *kk* ∩ *nat_Nseq* $^{-1}$ [{ *vv* }]
IN
bb := bool (*ss* ≠ {}) ||
ii := min (*ss* ∪ { *size* (*nat_Nseq*) })
END
END ;
...
END

Figure 2.8: System library nat_Nseq

Figure 2.9: The *B-Toolkit*'s architecture [9]

processes from parsing, type-checking to documenting all constructs. In order to maintain the state of a development, the toolkit has a "Construct manager" to act between its "Motif interface" and the *B-Platform*. The developers input via menus and buttons in the *Motif interface*, these are passed to the *Construct manager* where they are translated into series of goals for the *B-Platform*. In order to "discharge" these goals, the *B-Platform* invokes the appropriate rules in the *Toolkit binaries*. Any messages (such as, warning messages or error messages) will be passed in the opposite direction, through the *B-Platform* and the *Construct manager* to the *Motif interface* for the developers. These processes can be seen in Fig. 2.9. We will concentrate on how to write the rules to create the *Toolkit binaries*, which will help us to integrate probabilistic features into the *B-Method*.

For a construct (specification, refinement or implementation), its life-cycle is as follows:

Analysis. The analysing process involves *normalising*, *syntax checking* and *type checking*. In the normalising phase, the definitions are expanded; the construct is de-sugared from *AMN* to *GSL*. All processes are driven by proof rules. A proof rule is formed by constructing a goal and list of antecedents. The current goal is discharged by solving all these antecedents. Each antecedent becomes the sub-goal and the process is repeated.

For example, in order to de-sugar a *CHOICE* clause, the following proof rules can be used:

$$\frac{\begin{array}{l} Desugar(S) \ \wedge \\ MergeAndPush \quad \text{/*Merge the result with the top of the stack and push the after-merged result to the stack*/} \end{array}}{Desugar(\textbf{CHOICE } S \textbf{ END})} \tag{2.10}$$

$$\frac{\begin{array}{l} Desugar(T) \ \wedge \\ Push \ \wedge \quad \text{/* Push the result on the stack */} \\ Desugar(\textbf{CHOICE } S \textbf{ END}) \end{array}}{Desugar(\textbf{CHOICE } S \textbf{ OR } T \textbf{ END})} \tag{2.11}$$

The order of these proof rules is also very important. The rules are chosen from bottom up. Starting with the original goal to de-sugaring a *CHOICE* clause (which matches with the goal of the second rule), the theorem prover (which is used to discharge the goal) has to discharge a sub-goal (which is the first antecedent of (2.11)), in order to de-sugar T. Then the theorem prover pushes the result onto the stack (the second antecedent). It discharges the last sub-goal which is to de-sugar the clause "*CHOICE* S *END*" using the first proof rule (2.10). Every process in the *B-Toolkit* is driven by similar sets of rules.

After normalisation, the construct undergoes the syntax checking stage. The *B-Toolkit* checks that the clauses used in the construct are consistent with the rules of Abstract Machine. After this stage, the construct needs to be type-checked. Every variable or constant in *B* needs to be type-consistent. Types in *B* are defined by set-inclusion properties, e.g. $var \in \mathbb{N}$ or $const \in \mathbb{N}_1$. At the end of this analysing process, a concise version of the construct is generated and, from this point onward, other processes will work with this concise form of the construct. This means that the construct no longer needs to be parsed or type-checked again (unless the source code is changed).

Details about the proof rules' syntax and semantics can be found in [1, 2].

Proof-obligation generation. After the analysis phase, proof obligations can be generated for the analysed constructs. Depending on the type of the construct (specification, refinement or implementation), proof obligations can be generated accordingly. For example, the following specification

MACHINE MachineName(x)
CONSTRAINTS P
CONSTANTS c
PROPERTIES Q
VARIABLES v
INVARIANT R
INITIALISATION T
OPERATIONS
z ⟵ OpName ≙ **PRE** L **THEN** S **END**;
...
END

yields the following proof obligations:

$$
\begin{array}{lrcl}
1. & & & \exists\, x \cdot P \\
2. & P & \Rightarrow & \exists\, c \cdot Q \\
3. & P \wedge Q & \Rightarrow & \exists\, v \cdot R \\
4. & P \wedge Q & \Rightarrow & [T]\, R \\
5. & P \wedge Q \wedge R \wedge L & \Rightarrow & [S]\, R
\end{array}
$$

The first three obligations are for the existence of the machine parameters, constants and variables, respectively. This is purely for the consistency of the context information (this is the reason why they are called context proof-obligations). If the context information is inconsistent, the specification will not be implementable. The last two obligations are for the initial establishment of the invariant and maintenance of such an invariant.

We consider the most general case of refinement (both data and algorithmic) for the above machine as follows:

REFINEMENT RefinementName
REFINES MachineName
VARIABLES w
INVARIANT N
INITIALISATION U
OPERATIONS
z $\longleftarrow$ OpName $\widehat{=}$ **PRE** M **THEN** V **END**;
...
END

which yields the following proof obligations:

$$
\begin{array}{ll}
1. & \exists(v, w) \cdot (R \wedge N) \\
2. & [U]\, \neg([T]\, \neg N) \\
3. & \forall(v, w) \cdot (R \wedge N \wedge L \Rightarrow M) \\
4. & \forall(v, w) \cdot (R \wedge N \wedge L \Rightarrow [V']\, \neg([T]\, \neg(N \wedge z = z'))
\end{array}
$$

In the fourth proof obligation, V' is the substitution in which all of the occurrences of output z have been replaced by z'. The first obligation deals with the consistency of the variables and invariant. The second obligation deals with the establishment of the invariant by the initialisation. The third obligation states that whenever the specification's operation can be invoked, the corresponding operation

in the refinement can also be invoked. The fourth obligation says that the refinement's operation establishes the states where the specification's operation cannot fail to establish the invariant N, while also preserving the value of the output z.

The complete set of rules for proof obligation generation can be found in [4, 38].

Loops. A special construct in *B* is the loop. The *B-Method* supports loops but only within implementation constructs. If the developers want to have loops in the specification or refinement constructs, loops will be written in their abstractions and later refined to the actual loops, which will require a proof that they do in fact refine their abstractions. The semantics of loops can be given by a least-fixed point formula [4], but here we only consider the proof obligations for loops which are based on the *Invariant/Variant Theorem* [19, 4]. The idea is to denote loops with an invariant predicate I and a variant V (which is a natural number valued expression of the state). We have the following theorem that provides sufficient conditions for proving the correctness of loops.

Theorem 1 *Assume we have the following WHILE loop (which we denote "loop")*

WHILE G ***DO***
S
INVARIANT I
VARIANT V
END ,

and we want to prove $[loop]\ Q$ *where* Q *is an arbitrary post-condition. If we have the following conditions:*

- *the invariant* I *is maintained by the execution of the body of loop, i.e.*

$$G \wedge I \Rightarrow [S]\, I\ ,\ and$$

- *on termination, the post-condition* Q *is established, i.e.*

$$\neg G \wedge I \Rightarrow Q\ ,\ and$$

- *the variant* V *is natural number, i.e.*

$$G \wedge I \Rightarrow V \in \mathbb{N}\ ,\ and$$

- *the variant* V *decreases for every iteration of the loop, i.e.*

$$\forall N.(G \wedge I \wedge (V = N) \Rightarrow [S]\,(V < N))\ ,$$

then $I \Rightarrow [loop]\ Q$*, i.e. we can prove* I *instead of* $[loop]\ Q$*.*

Later, we extend the above theorem to cover probability-one termination and general probabilistic loops.

Proof of obligations. The *B-Toolkit* provides three different provers for discharging the obligations generated in the earlier steps. The developers can use a fully automatic prover (which uses the standard library rules) called *AutoProver*, a manual prover called *BToolProver* or a semi-automatic prover called *InterProver*. With the *BToolProver*, extra proof rules can be added, in order to help with proving the obligations. Discharging proof obligations guarantees that the construct is consistent (in the case of specifications), and the refinement relationship is correct (in the case of refinements and implementations).

Animation. The developers can animate any specification construct. The purpose of animating is to validate that the specification is what the users want (verifying against requirements). Since a specification is the first step in a development, it is very important to have a correct specification [31, 36].

Translation. For the implementation construct, the translation to executable code is straightforward (by simply clicking a button). The translation process is also controlled by proof rules. The produced executable code is the code that provides the functionality of the specified system. However, it still needs an interface in order to be run properly.

Interface generation. In order to run the code generated by the translation process, an interface construct needs to be added to the development. The interface construct contains the name of the operations that the developers want to generate the interface for.

Documentation. Once analysed, all the constructs in the *B-Toolkit* can be documented, and this process is also driven by proof rules. This includes the obligations and their proofs.

The nature of the *B-Toolkit*—that is built on top of a theorem prover— has the advantage that it can be extended easily by adding new rules. Once the syntax and semantics of the proof rules have been studied, adding new features to the *B-Toolkit* is just a matter of writing the corresponding proof rules.

2.5 The Probabilistic Generalised Substitution Language

The (now called) "standard" *GSL* acts over predicates, and its substitutions can describe both the non-deterministic and deterministic behaviours of software. Non-deterministic behaviour allows *GSL* to be used as a specification language, where implementation issues can be deferred to later stages via refinement. A specification is usually very simple and easy to understand, in order to be easily compared with the users' requirements.

Even so, the standard *GSL* cannot be used to describe systems with probabilistic non-deterministic behaviours. For example, a system containing sensors which have certain probabilities of failure would not be specified. Other limitations include the inability to specify and implement randomised algorithms. In other words, the standard *GSL* cannot describe the behaviour of substitutions such as: "do S with some probability and do T otherwise". The closest it can get to describing this is to use a non-deterministic substitution such as

$$S \quad [\!] \quad T\,,$$

which means that either S or T is performed but the information about probability is ignored. Morgan's proposal of *pGSL* [45] provides a solution to this problem.

In this section, we first review some elementary concepts from probability theory; then we look at the syntax of the new language; then we present the semantics of *pGSL*; and finally, we consider some examples of *pGSL*.

2.5.1 Elementary Probability Theory

Probability theory [18] provides the following concepts which are sufficient for our purpose. The principal concepts we need are distribution and expected value.

Experiment: Any process of observation or measurement.

Outcomes: The results obtained from an experiment.

Sample space: The set of all possible outcomes of an experiment.

Event: A subset of the sample space.

Probability distribution (discrete): A normalised function from the sample space to $[0, 1]$ giving the probability of each outcome.

Random variable: Any function from the sample space into the reals.

Characteristic function: The *characteristic function* of an event is a random variable that takes value 1 for outcomes in the event, and 0 otherwise.

Expected value (discrete): If f is a bounded random variable and μ is a discrete distribution, both over sample space S, then the expected value of f over μ is defined:

$$\sum_{s \in S} f(s) \times \mu(s) \ .$$

As a consequence of the above definitions, it is easy to see that the expected value of a characteristic function over a distribution is equal to the probability assigned to its underlying set by that distribution.

2.5.2 *pGSL* Syntax

The probabilistic choice substitution is the only extra substitution in *pGSL*. It has the following form

$$S \quad {}_p\!\oplus \quad T \ . \tag{2.12}$$

The meaning of the substitution is that the left branch S is chosen with probability p and the right branch T is chosen with probability $1 - p$. The probability p can be a function of the state that takes the value between 0 and 1 (inclusive). So the language of *pGSL* now involves both non-deterministic and probabilistic substitutions. It proves to be a challenging task to have both kinds of substitutions in the language. The language is now capable of modelling systems with probabilistic

and non-deterministic behaviours, i.e. it can be used as a specification language for probabilistic systems.

As a first example of a program written in *pGSL*, consider

$$x := 1 \quad {}_{\frac{1}{2}}\!\oplus \quad x := 2\,, \tag{2.13}$$

which sets x to either 1 or 2 with equal probability (one-half). As a more complex example, consider a program with both probabilistic and non-deterministic substitutions.

$$\Big(x := 1 \;{}_{\frac{1}{3}}\!\oplus\; x := 2\Big) \quad [\!] \quad \Big(x := 1 \;{}_{\frac{2}{3}}\!\oplus\; x := 2\Big)\,. \tag{2.14}$$

The meaning of this program is either assign 1 to x with probability $\frac{1}{3}$ and assign 2 to x otherwise, or to assign 1 to x with probability $\frac{2}{3}$ and assign 2 to x otherwise. Overall, the effect of Prog. (2.14) is to set x to 1 with a probability between $\frac{1}{3}$ and $\frac{2}{3}$, and otherwise to set x to 2.

2.5.3 *pGSL* Semantics

In order to accommodate the extension, *pGSL* uses real-valued expressions instead of Boolean-valued expressions for its "predicates", which are now called "expectations": the numbers represent "expected values" rather than the normal predicate that either does, or does not hold. In other words, we replace certainty by probability. The notion of *weakest precondition* in *GSL* gives the (weakest) condition under which a certain program is guaranteed to establish a post-condition Q. There is a similar notion in *pGSL* called *weakest pre-expectation*, which gives the lower bound (evaluated in the initial state) of a post-expectation after executing the program. Imagine that a program S is executed from the same initial state many times: the expected value of post-expectation B in the resulting final states is then to be "at least" [3] the actual value of the pre-expectation A in the initial state.

As an example, consider the program Prog. (2.13): what can we say about the value of the expression x^2? It can be either 1 or 4 depending on which branch the execution takes, but moreover, we can say that (in a long run) the expected value of x^2 is

$$\frac{1}{2} \times 1^2 \;+\; \frac{1}{2} \times 2^2 \quad = \quad \frac{5}{2}\,.$$

[3] The reason why it is "at least" rather than "equal to" will be explained later.

With this intuition, we start with two definitions:

$$\begin{array}{lcl} [\![x := E]\!]\, B & \widehat{=} & B \text{ with } x \text{ replaced everywhere by } E,\,[4] \\ [\![S \;{}_p\!\oplus\; T]\!]\, B & \widehat{=} & \begin{array}{rcl} p & \times & [\![S]\!]\, B \\ + \;\; (1-p) & \times & [\![T]\!]\, B\,. \end{array} \end{array}$$

In the above definitions, we use double brackets $[\![\cdot]\!]$ to distinguish the substitutions of probabilistic programs from standard programs ($[\cdot]$). Later, we will argue that there are not many differences between them and the notation can be merged together to avoid confusion.

What about questions such as "What is the probability that $x \leq 1$ after the execution of Prog. (2.13)?" It is easy to see that the probability is exactly $\frac{1}{2}$ (the probability of executing the left-branch), but how can we formally reason about this? We introduce the notion of an embedded predicate $\langle P \rangle$. We define $\langle \mathsf{true} \rangle$ to be 1 and $\langle \mathsf{false} \rangle$ to be 0, so that $\langle P \rangle$ is just the probability that a given predicate P holds[5]. Trivially, if P is false, it holds with probability 0 and if P is true, it holds with probability 1. Returning to the question above, it turns out that the answer is just $[\![\mathit{Prog}.\ (2.13)]\!]\, \langle x \leq 1 \rangle$. We can calculate the probability as follows:

$$\left[\!\!\left[x := 1 \;{}_{\frac{1}{2}}\!\oplus\; x := 2 \right]\!\!\right] \langle x \leq 1 \rangle$$

$$\begin{array}{lcll} \equiv & \begin{array}{rcl} \frac{1}{2} & \times & [\![x := 1]\!]\, \langle x \leq 1 \rangle \\ + \;\; (1 - \frac{1}{2}) & \times & [\![x := 2]\!]\, \langle x \leq 1 \rangle \end{array} & \text{probabilistic choice substitution} \\ \equiv & \frac{1}{2} \times \langle 1 \leq 1 \rangle + \frac{1}{2} \times \langle 2 \leq 1 \rangle & \text{simple substitution} \\ \equiv & \frac{1}{2} \times 1 + \frac{1}{2} \times 0 & \text{logic and embedded predicate} \\ \equiv & \frac{1}{2} & \text{arithmetic} \end{array}$$

Implication-like relations between expectations are defined as follows:

$$\begin{array}{lcl} A \Rrightarrow B & \widehat{=} & A \text{ is everywhere no more than } B \\ A \equiv B & \widehat{=} & A \text{ is everywhere equal to } B \\ A \Lleftarrow B & \widehat{=} & A \text{ is everywhere no less than } B \end{array}$$

[4] Here, we only replace free occurrences of x in the substitution. If necessary, the variables will be renamed to avoid name clashes.

[5] In fact, $\langle P \rangle$ is the *characteristic function* of predicate P.

These relations are used to give the definition for (algorithmic) refinement between programs in *pGSL*. Program S is refined by program T if, for all expectations B, we have

$$[S]\,B \quad \Rrightarrow \quad [T]\,B\,.$$

We will extend the *B-Method* with the probabilistic choice substitution and use this definition of refinement in our new framework.

Even though we have used the double brackets ($[\![\cdot]\!]$) for probabilistic substitution, there is little distinction to make when the actual program is standard (i.e., when it contains no probabilistic choice substitution). In fact, the calculation for standard programs in a probabilistic framework is effectively the same, as the following lemma shows.

Lemma 1 *If S is a standard and feasible program — that is, one which contains no probabilistic choice and is non-miraculous — then for any post-condition Q, we have*

$$[\![S]\!]\,\langle Q\rangle \quad \equiv \quad \langle [S]\,Q\rangle\,.$$

The lemma states that it is irrelevant for one to calculate the pre-expectation or the precondition (and then use an embedding to convert into a real number), in the case of a standard program with respect to a standard precondition. The intuition of the lemma is that the standard implication and the implication-like relation in *pGSL* agrees for standard predicates, i.e.

$$P \;\Rightarrow\; Q \qquad \text{exactly when} \qquad \langle P\rangle \;\Rrightarrow\; \langle Q\rangle\,.$$

More details and the intuitive meaning of this lemma can be found elsewhere [41].

From now on, we will use the substitution ($[\cdot]$) for probabilistic substitutions. The distinction between probabilistic and standard substitution will be based on the nature of the program itself with respect to predicates or expectations.

The semantics of probabilistic substitutions needs to be changed according to the replacement of expectations by predicates. The summary for the semantics of *pGSL* can be seen in Fig. 2.10 on the next page.

For the healthiness condition and algebraic properties of *pGSL*, the reader is referred to other publications [49].

We now consider the Prog. (2.14). We can ask the same question "What is the probability that $x \leq 1$ after the execution of Prog. (2.14)?" There are no exact

$[x := E]\, B$	The expectation by replacing all free occurrences of x by E in B.
$[P \mid S]\, B$	$P \times [S]\, B$.
$[S \parallel T]\, B$	$[S]\, B \ \min\ [T]\, B$.
$[S \ {}_p\!\oplus\ T]\, B$	$p \times [S]\, B \ + \ (1-p) \times [T]\, B$.
$[S; T]\, B$	$[S]\,([T]\, B)$.
$[P \Longrightarrow S]B$	$1/P \ \times \ [S]\, B.$
[skip] B	B .
$[@\, x \cdot S]\, B$	$\sqcap x \cdot ([S]\, B)$.

- P is a predicate;
- B is an expectation;
- S, T are probabilistic generalised substitutions;
- x is a variable (or a vector of variables);
- $\sqcap x \cdot (E)$ is the minimum of E for all x.
- We also assume that $\infty \times n = \infty$ for any positive number n.

Figure 2.10: *pGSL* — the *pGSL* semantics [45]

numbers that answer the question (because of the non-determinism). To be precise, we have to re-phrase our question as: "What is the *least* guaranteed probability that $x \leq 1$ after the execution of Prog. (2.14)?" If we apply the semantics given in Fig. 2.10, we calculate the answer as follows:

$$\Big[\Big(x := 1 \ {}_{\frac{1}{3}}\!\oplus\ x := 2\Big) \quad \parallel \quad \Big(x := 1 \ {}_{\frac{2}{3}}\!\oplus\ x := 2\Big)\Big]\, \langle x \leq 1 \rangle$$

$\equiv$ non-deterministic substitution

$$\min \begin{array}{l} \Big[x := 1 \ {}_{\frac{1}{3}}\!\oplus\ x := 2\Big]\, \langle x \leq 1 \rangle \\ \Big[x := 1 \ {}_{\frac{2}{3}}\!\oplus\ x := 2\Big]\, \langle x \leq 1 \rangle \end{array}$$

$$\equiv \qquad \text{probabilistic choice substitutions}$$

$$\min \begin{pmatrix} \frac{1}{3} & \times & [x := 1]\,\langle x \leq 1\rangle \\ + & (1-\frac{1}{3}) & \times & [x := 2]\,\langle x \leq 1\rangle \end{pmatrix} \begin{pmatrix} \frac{2}{3} & \times & [x := 1]\,\langle x \leq 1\rangle \\ + & (1-\frac{2}{3}) & \times & [x := 2]\,\langle x \leq 1\rangle \end{pmatrix}$$

$$\equiv \qquad \text{simple substitutions}$$

$$\min \begin{pmatrix} \frac{1}{3} & \times & \langle 1 \leq 1\rangle \\ + & (1-\frac{1}{3}) & \times & \langle 2 \leq 1\rangle \end{pmatrix} \begin{pmatrix} \frac{2}{3} & \times & \langle 1 \leq 1\rangle \\ + & (1-\frac{2}{3}) & \times & \langle 2 \leq 1\rangle \end{pmatrix}$$

$$\equiv \quad \left(\tfrac{1}{3}\times 1 + \tfrac{2}{3}\times 0\right) \quad \min \quad \left(\tfrac{2}{3}\times 1 + \tfrac{1}{3}\times 0\right) \qquad \text{logic and embedded predicates}$$

$$\equiv \quad \tfrac{1}{3} \quad \min \quad \tfrac{2}{3} \qquad \text{arithmetic}$$

$$\equiv \quad \tfrac{1}{3}\,. \qquad \text{arithmetic of } \min$$

So we conclude that the probability of Prog. (2.14) to establish $x \leq 1$ is *at least* $\frac{1}{3}$, which is reasonable since the program sets x to 1 with a probability between $\frac{1}{3}$ and $\frac{2}{3}$. Because of the decision to have "$\min$" for the semantics of the non-deterministic substitution, the meaning of the *weakest pre-expectation* is *at least* rather than *equal to*.

2.6 Summary

With the introduction of probability into *GSL* we obtain the new language *pGSL*. *B* can be extended to cover probabilistic programs (programs with probabilistic properties), from specifications to implementations. There is a need to have a tool to support the method. The fact that the *B-Toolkit* is built on top of a theorem prover reduces the work needed in order to integrate probabilistic features. We simply need to construct the appropriate proof rules.

As a side effect of extending the *B-Toolkit*, we want to retain the original

method (and hence the *B-Toolkit*) as much as possible. We want to have a simple extension of the method, whose application is sufficiently practical enough, but also easy to build into the *B-Toolkit*. In a field where the full theoretical treatment of both probabilistic choice and non-deterministic choice seems prohibitively complex, we want to use practical problems to guide us towards the issues that are most important.

The integration of probability into the *B-Method* results in a new method, which can be applied to a wide range of problems: firstly for systems with almost-certain termination (Chap. 3); secondly, with probabilistic invariant for probabilistic machines (Chap. 4); and thirdly, the concept of refinement within the new framework, which is retained (and explored in Chap. 5 and Chap. 6).

Chapter 3

Almost-certain Termination

3.1 Demonic- versus Probabilistic Termination

In some systems, we cannot be certain that the system will achieve a particular result, but when we calculate the probability of obtaining such a result, it is actually 1. The simplest example is tossing a coin. If the coin is flipped often enough, then "almost certainly" it will turn up heads. A more complex example is tossing two coins: with similar reasoning, we can say that both coins will eventually be the same with probability one. This kind of reasoning is often used in many existing algorithms in distributed systems. It is an efficient way to break symmetry when compared to normal deterministic algorithms.

Standard (non-probabilistic) B is currently not able to specify such systems or indeed prove that they terminate with probability one. It can describe non-determinism, for example, a coin that either turns up heads or tails. However, it is not good enough to say that flipping the coin often enough will turn up heads, since the non-deterministic choice allows the coin to turn up tails every time (Fig. 3.1 on the next page) [1].

With the introduction of the "abstract probabilistic choice" operator ($\oplus$), one can specify an abstract coin such as in Fig. 3.2 on the following page. In this case,

[1] Here, we consider what is called demonically non-deterministic choice. There is also "angelic" non-deterministic choice, which is used to specify the non-deterministic behaviour *angelically*, i.e. a choice that can do either branch, but of which you can assume it always picks the better branch. From now on, we assume that when talking about non-determinism, we are talking about demonically non-deterministic choice.

$coin := Tail$;
WHILE $coin = Tail$ **DO**
  $coin := Tail \ [] \ coin := Head$
END

Figure 3.1: A demonic coin

the coin also either comes up heads or tails but with some unknown probability. (Later, we specify what the constraints for this probability are.) It is called *abstract* probabilistic choice since we do not specify the exact probability of the choice. The termination of the loop in the program in Fig. 3.2 happens with probability one no matter which probability (provided not 0) is used for the coin (with some exceptions). This is an important factor in simplifying the implementation of this kind of "termination with probability one" algorithms.

$coin := Tail$;
WHILE $coin = Tail$ **DO**
  $coin := Tail \oplus coin := Head$
END

Figure 3.2: An abstract coin

We want to give a formal notation and argument of how to express and reason about the termination with probability one without significant changes to the current *B* language.

In this chapter, we first study the zero-one law for probabilistic loops [41]; we then extend the *GSL* syntax to accommodate the *abstract probabilistic choice* operator $\oplus$ and look at its corresponding *AMN*. We give the new proof obligations for loops so that they are correct with respect to the new operator; we summarise the necessary changes that have been made to the *B-Toolkit*; finally, we study two examples, namely Root Contention (in the FireWire protocol [27, 28]) and Rabin's Choice-Coordination algorithm [57], which have been developed and proved using the modified *B-Toolkit*.

3.2 A Probabilistic *Zero-one* Law for Loops

In this section, we review the almost-certain property discussed in more details in [41] and extract the rules required for the zero-one law.

3.2.1 Almost-Certain Correctness

Recall that in *pGSL* we usually deal with conclusions of the form

$$A \quad \Rrightarrow \quad [S]\, B \ ,$$

which can be interpreted as the "the expected value of B in the final state is at least the (actual) value of A in the original state". In the case that the post-expectation is standard, i.e. of the form $\langle Q \rangle$, where Q is a standard predicate, $[S]\,\langle Q \rangle$ is the greatest guaranteed probability that predicate Q will hold after the execution of the program S [49]. The post-condition is established "almost-certainly" when the probability of establishing that post-condition is one. We start with the following definition [41].

Definition 1 *Probabilistic program S establishes post-condition Q* almost-certainly *from precondition P when the following condition holds:*

$$\langle P \rangle \quad \Rrightarrow \quad [S]\,\langle Q \rangle \ .$$

The intuitive meaning of the definition is that when P does not hold, $\langle P \rangle$ is 0, which makes the inequality hold trivially. When P holds, i.e. $\langle P \rangle$ is 1 and then we must have $[S]\,\langle Q \rangle$ is 1 also, which gives the above meaning of almost-certain correctness.

The standard *GSL* logic is not defined for probabilistic choices, both abstract ($\oplus$) and concrete ($_p\oplus$). The best approximation one can get for probabilistic choice in *GSL* is to use non-deterministic substitutions, but even with that, *GSL* fails to prove the almost-certain correctness in most cases. *GSL*'s approximation in some cases is too abstract, hence the information about termination will not be preserved and should not be expected to be so. A trivial example is the program in Fig. 3.1 on the facing page. It is the non-deterministic approximation of the program in Fig. 3.2 on the preceding page. However, while we know that (with some informal reasoning) the program in Fig. 3.2 terminates with probability one, we cannot prove

$coin := Tail$;
WHILE $coin = Tail$ **DO**
  $coin := Tail \ {}_{0.5}\oplus \ coin := Head$
END

Figure 3.3: A probabilistic coin

termination of the program in Fig. 3.1. In the next section, we will look at the failure of the standard variant rule for loop when proving termination of this kind.

3.2.2 The Failure of the Standard Variant Rule

As we stated in the last section, the standard variant rule is not strong enough to prove the almost-certain termination of a probabilistic loop (if we use a non-deterministic approximation for the probabilistic choice). Consider the program in Fig. 3.3. To avoid termination, the left branch (i.e. $coin := Tail$) must be chosen in every iteration of the loop. The probability of this happening n times in succession is $\frac{1}{2^n}$. Consider the case where the left branch is chosen "forever". The probability for that to happen therefore is $\frac{1}{2^\infty}$, and is effectively zero. So the probability that the right branch (i.e. $coin := Head$) is chosen eventually —and hence that the loop terminates— is essentially one.

Consider also the more abstract program in Fig. 3.2 on page 40, where the concrete choice (${}_{0.5}\oplus$) is replaced by abstract choice ($\oplus$). Imagine that the abstract probabilistic choice is in fact a probabilistic choice with unknown probability p. With similar reasoning, the probability for the program avoiding termination is p^∞. So as long as $p \neq 1$, the program will be guaranteed to terminate with probability one. Later, we will formalise the "proper" condition on p.

A standard program that can be used to describe the behaviour of both programs in Fig. 3.2 and Fig. 3.3 is the non-deterministic program in Fig. 3.1 on page 40. However, the standard *GSL* variant rule fails to prove the termination of such programs: there are no variants that are guaranteed to decrease for every iteration of the loop, since indeed it does not necessarily terminate at all.

The failure of finding a variant for the loop indicates that we need to reason about these probabilistic programs in a *pGSL* context. We need to strengthen the

standard variant rule to concentrate on properties of the probabilistic choice (especially in the abstract case) with respect to almost-certain correctness. With the introduction of the "zero-one" law in the next section, we will be able to reason about this kind of correctness, which we are not able to in the standard semantics of *GSL*.

3.2.3 Termination of Probabilistic Loops

In this section, we review the definition of the demonic retraction of probabilistic programs, then based on that we give the *zero-one law* for loops.

In the previous section, we said that the program in Fig. 3.1 on page 40 is a standard approximation for the programs in Fig. 3.2 on page 40 and Fig. 3.3 on the preceding page. We will now take a closer look at this claim. Recall from Sec. 3.2.1 that $[S]\langle Q\rangle$ is the greatest guaranteed probability that Q is established by the execution of S. If we want to find a standard approximation of S, then we want to have a program S_d which is a standard program and is an abstraction of S, i.e. $S_d \sqsubseteq S$. By the definition of refinement (Sec. 2.5.3), it would be necessary to have:

$$[S_d]\langle Q\rangle \quad \Rrightarrow \quad [S]\langle Q\rangle\,, \quad \text{for all predicate } Q\,. \tag{3.1}$$

On the other hand, the left-hand side of (3.1) can have values only either 0 or 1 since it is a standard program applied to a standard post-condition, and hence it is trivially true that (3.1) is equivalent to

$$[S_d]\langle Q\rangle \quad \Rrightarrow \quad \lfloor [S]\langle Q\rangle \rfloor\,, \tag{3.2}$$

where $\lfloor\cdot\rfloor$ is the mathematical *floor* function [2].

Recall from Def. 1 that the condition for almost-certain correctness for a probabilistic program S to establish post-condition Q from precondition P is

$$\langle P\rangle \quad \Rrightarrow \quad [S]\langle Q\rangle\,.$$

Since $\langle P\rangle$ has the value either 0 or 1, it is equivalent to

$$\langle P\rangle \quad \Rrightarrow \quad \lfloor [S]\langle Q\rangle \rfloor\,. \tag{3.3}$$

[2] The *floor* of a real number is the greatest integer that does not exceed it.

Hence, for syntactic convenience, we make the following definition

$$\llbracket S \rrbracket\, Q \quad \widehat{=} \quad ([S]\,\langle Q\rangle = 1)\ .$$

Even though S can be probabilistic, $\llbracket S \rrbracket\, Q$ always has a Boolean value, i.e. it is a predicate. Therefore, we can rewrite Fig. 3.3 on the preceding page, and state the following definition.

Definition 2 *Let S be a probabilistic program and let P and Q be predicates of the state. We say that Q is almost-certainly established under execution of S from any initial state satisfying P if*

$$P \quad \Rightarrow \quad \llbracket S \rrbracket\, Q\ . \tag{3.4}$$

(Notice we use $\Rightarrow$ to indicate that (3.4) is in fact a standard predicate).

We call $\llbracket S \rrbracket$ the *demonic retraction* of $[S]$.

With the definition of demonic retraction, the zero-one law for probabilistic loops can be defined as in the following lemma.

Lemma 2 *Let*

WHILE G ***DO***
  S
INVARIANT I
END

be a loop (denoted by "loop") with invariant I being a predicate. If there exists a number δ strictly greater than zero and we have that

$$I \wedge G \quad \Rightarrow \quad \llbracket S \rrbracket\, I \tag{3.5}$$

$$\textit{and} \quad \delta \times \langle I\rangle \quad \Rrightarrow \quad [\mathit{loop}]\,\langle \mathit{true}\rangle \tag{3.6}$$

both hold, then in fact

$$\langle I\rangle \quad \Rightarrow \quad [\mathit{loop}]\,\langle I \wedge \neg G\rangle\ .$$

The intuitive meaning and proof of the above lemma can be found elsewhere [41, 44]. Here we concentrate on the application of the above lemma and define the probabilistic variant rule for probabilistic loops.

3.2.4 Probabilistic Variant Rule for Probabilistic Loops

The purpose of the probabilistic variant rule for loops is to reason about almost-certain termination within the scope of standard predicates. In this section, we look at some lemmas that support Boolean reasoning for probabilistic programs, and then give the probabilistic variant rule.

In the previous section, we use demonic retraction to define the zero-one law for loops. The property that makes demonic retraction so attractive to us is its distributivity. The demonic retraction distributes just as the normal substitution ($[\cdot]$) does, but it is also defined for probabilistic choice ($_p\oplus$). Moreover, the result of demonic retraction is still in the Boolean domain, which means that the overall reasoning is purely within the Boolean logic, as the following lemma shows.

Lemma 3 *Demonic retraction has the same* [3] *distributivity properties as the standard substitution* $[\cdot]$*, and extends it as follows: if*

$$0 < p < 1$$

then

$$\llbracket S \ {}_p\!\oplus \ T \rrbracket\, Q \quad \equiv \quad \llbracket S \rrbracket\, Q \ \wedge \ \llbracket T \rrbracket\, Q \,.$$

Proof. For $0 < p < 1$, we have

$$\begin{array}{lll} & \llbracket S \ {}_p\!\oplus \ T \rrbracket\, Q & \\ \equiv & [S \ {}_p\!\oplus \ T]\, \langle Q \rangle \ = \ 1 & \text{demonic retraction definition} \\ \equiv & p \times [S]\, \langle Q \rangle \ + \ (1-p) \times [T]\, \langle Q \rangle \ = \ 1 \,. & \text{probabilistic choice} \end{array}$$

Since both $[S]\,\langle Q\rangle$ and $[T]\,\langle Q\rangle$ are between 0 and 1 (inclusive), we have

$$p \times [S]\, \langle Q \rangle \ + \ (1-p) \times [T]\, \langle Q \rangle \ = \ 1$$

if and only if

$$[S]\, \langle Q \rangle \ = \ 1 \quad \text{and} \quad [T]\, \langle Q \rangle \ = \ 1,$$

or equivalently

$$\llbracket S \rrbracket\, Q \text{ and } \llbracket T \rrbracket\, Q \text{ both hold.}$$

[3] An exception is that *IF* $\cdots$ *END* must be treated as a whole.

As for the substitution $[\cdot]$, there are no rules for distribution of $[\![\cdot]\!]$ through *WHILE* loops; instead it is approximated by the invariant/variant rules (see Sec. 2.4). Based on Lem. 2, we can build the invariant/variant rules for probabilistic loops.

With this definition, one can see that the program in Fig. 3.1 on page 40 is the demonic retraction of the program in Fig. 3.3 on page 42.

Before looking at the probabilistic variant rule, we need to introduce some definitions as follows.

Definition 3 *Let S be a probabilistic program in* pGSL*, and let P and Q be predicates of the state. We say that Q has "some chance" of being established under execution of S from any initial state satisfying P if*

$$\langle P \rangle \quad \Rrightarrow \quad \lceil [S] \langle Q \rangle \rceil \,, \tag{3.7}$$

where $\lceil \cdot \rceil$ is the mathematical ceiling function[4]*. As before, we make a definition for syntactic convenience:*

$$[\![S]\!] Q \quad \widehat{=} \quad ([S] \langle Q \rangle \neq 0) \,.$$

Furthermore, $[\![S]\!] Q$ is a predicate even though S can be probabilistic. And we can rewrite (3.7) as follows:

$$P \quad \Rightarrow \quad [\![S]\!] Q \,.$$

We call $[\![S]\!]$ the "angelic retraction" of $[S]$.

It turns out that "some chance" is not strong enough and we need to specify how small the chance can be with the following definitions.

Definition 4 *A probabilistic program S is "definite" if there exists a positive* constant Δ *such that, for all post-conditions Q, if S establishes Q with non-zero probability, then that probability is at least Δ. In other words*

$$\Delta \times \lceil [S] \langle Q \rangle \rceil \quad \Rrightarrow \quad [S] \langle Q \rangle \,. \tag{3.8}$$

More information on angelic retraction and definiteness can be found elsewhere [41].

Similarly to demonic retraction, we consider the distributivity of angelic retraction, and it turns out that a similar result can be achieved, as the following lemma shows.

[4] The *ceiling* of a real number is the least integer that it does not exceed.

Lemma 4 *Angelic retraction has the same*[5] *distributivity properties as the standard substitution* $[\cdot]$*, and extends it as follows: if*

$$0 < p < 1$$

then

$$[\![S \ {}_p\!\oplus \ T]\!]\, Q \quad \equiv \quad [\![S]\!]\, Q \ \vee \ [\![T]\!]\, Q \,.$$

There is also an extra condition for the distributivity of $[\![\cdot]\!]$ into unbounded choice, which is related to the definition *uniformly definite*. Full details can be found in [41]. The definition is required to make sure that an infinite family of non-zero probabilities does not have infimum zero. An important point to notice (similar to the case of demonic retraction) is that the distributivity converts probabilistic choice into Boolean disjunction, which again allows the overall reasoning to stay within Boolean logic. Similarly, there are no distribution rules through *WHILE* loops.

With the introduction of angelic retraction, we can now reformulate the second rule (3.6) of Lem. 2 as the following probabilistic variant rule.

Lemma 5 *Let*

WHILE G ***DO***
  S
INVARIANT I
VARIANT V
END

be a loop (denoted by "loop") with invariant I *being a predicate, and variant* V *being an integer-valued expression over the state. If we have*

- V *is bounded above and below, i.e. there are integer constants* L, U *such that*

$$I \wedge G \quad \Rightarrow \quad L \leq V \leq U \,; \tag{3.9}$$

- V *has some chance of decreasing, i.e. for any* N*, we have*

$$I \wedge G \wedge (V = N) \quad \Rightarrow \quad [\![S]\!] (V < N) \,; \ \textit{and} \tag{3.10}$$

- S *is definite,*

[5] Again we insist that *IF* $\cdots$ *END* be treated as a whole.

then there is a strictly positive δ such that

$$\delta \times \langle I \rangle \quad \Rrightarrow \quad [\mathit{loop}]\, \langle \mathit{true} \rangle \tag{3.11}$$

as required for Lem. 2.

Proof. Assume the loop has not terminated yet, i.e. $I \wedge G$ holds in the current state. From (3.9), the probability of termination from the current state is at least the probability of $U - L + 1$ successive decreases of V. However, from (3.10), we know that for each iteration, the probability of V decreasing is non-zero, and from the third condition (S is definite), the probability is at least Δ for some positive constant Δ. Therefore the overall termination probability is no less than Δ^{U-L+1}.

So (3.11) is satisfied by taking δ to be Δ^{U-L+1}.

With the above lemma, the proof obligation rules for loops can now be expressed entirely in Boolean logic, and we will re-assemble these in the next section.

3.2.5 Proof Obligation Rules for Probabilistic Loops

Informally, the proof obligations for probabilistic loops in order to prove almost-certain correctness can be restated as follows:

1. The variant is bounded both above and below;
2. The body of the loop is *definite*.
3. The invariant is preserved. In this case, probabilistic choice is interpreted demonically, i.e. using $[\![\cdot]\!]$; and
4. The variant decreases (with non-zero probability), in this case, we interpret probabilistic choice angelically, i.e. using $[\![\cdot]\!]$.

The most important point to notice is that in the above summary the exact probability is not referred to at all. Formally, we have the following theorem about the proof rule for probabilistic loops.

Theorem 2 *Assume we have the following loop (denoted by "loop"):*

$$\begin{array}{l} \textbf{\textit{WHILE}}\ G\ \textbf{\textit{DO}} \\ \quad S \\ \textbf{\textit{INVARIANT}}\ I \\ \textbf{\textit{VARIANT}}\ V \\ \textbf{\textit{END}}\ , \end{array}$$

where S is definite. Moreover, let I be a predicate, let V be an integer-valued expression over the state space; and let L and U be constant integers (they can be expressions of the state space, which are invariant over the execution of the loop body). Let Q be the post-condition; then

$$\begin{aligned} \text{If} \quad & I \wedge G \quad \Rightarrow \quad [\![S]\!] I && (3.12) \\ \text{and} \quad & I \wedge G \quad \Rightarrow \quad L \leq V \leq U && (3.13) \\ \text{and, for all N,} \quad & I \wedge G \wedge (V = N) \quad \Rightarrow \quad [\![S]\!] (V < N) && (3.14) \\ \text{and} \quad & I \wedge \neg G \quad \Rightarrow \quad Q && (3.15) \\ \text{then} \quad & \langle I \rangle \quad \Rightarrow \quad [\mathit{loop}] \langle Q \rangle\ . && (3.16) \end{aligned}$$

In the above theorem, we said that the body of the loop S must be definite in order to make sure that the theorem holds. Fortunately, this can be assured by the following conditions [6]:

- For any probabilistic choice, ensure that the probability of the choice is "proper", i.e., there exists a constant ϵ strictly greater than 0, such that for every probability p in ${}_p\oplus$ satisfies $\epsilon < p < 1 - \epsilon$, every time the choice is executed, and

- Do not allow (nested) *WHILE*-loops in the loop bodies.

Proofs that these conditions guarantee the definiteness can be found elsewhere [41].

Moreover, in the distribution laws for probabilistic choice, it does not matter what the exact value of the probability (p) is, as long as p is "proper" (as in the first condition). So we can simply ignore p altogether and, instead, use the "abstract

[6] We will address these issues again in the next section about the implementation of almost-certain termination.

probabilistic choice" ($\oplus$) with the "quasi-distribution laws" as follows:

$$\lfloor\!| S \oplus T |\!\rfloor\, Q \quad \equiv \quad \lfloor\!| S |\!\rfloor\, Q \wedge \lfloor\!| T |\!\rfloor\, Q \tag{3.17}$$

$$\lceil\!| S \oplus T |\!\rceil\, Q \quad \equiv \quad \lceil\!| S |\!\rceil\, Q \vee \lceil\!| T |\!\rceil\, Q \tag{3.18}$$

The above laws give us the opportunity to postpone the (real) arithmetic reasoning and just concentrate on the almost-certain correctness, which can be reasoned entirely in Boolean logic. This will be an important factor for extending the *B-Toolkit* to support almost-certain correctness, which we will look at in detail in the next section.

3.3 Implementation of Almost-certain Termination

The introduction of the abstract probabilistic choice operator ($\oplus$) into *GSL* requires changes in *B*, where the constructs (specifications, refinements, implementations) are written in the *AMN*. (Only after analysis are programs translated into the more concise *GSL* form.) We introduce a construct *ACHOICE* into *AMN* for this purpose, so that

ACHOICE
  S
OR corresponds to $S \oplus T$.
  T
END

Since we have to prove that variants for loops are bounded above as well as below, a new clause *BOUND* is introduced to declare the upper bound of a variant. (By convention, the variant is always a natural number, so we can assume that the lower bound is zero.) Thus, a *WHILE*-loop is now

WHILE G **DO**
  S
INVARIANT I
VARIANT V
BOUND U
END .

Here, the type checker needs to be changed in order to check the type of the new clause *BOUND*. The expression representing the upper bound must be correctly typed as a natural number.

When dealing with the proof obligation generator for loops, there are two parts: proof obligations concerning partial correctness (if the loop terminates, then it is correct) and total correctness (the loop does in fact (almost-certainly) terminate). The rules are based on the Thm. 2 in Sec. 3.2.5, with an exception that, instead of having a specific lower bound for the variant V, we prove that the variant is a natural number (i.e. having an explicit 0 lower bound).

Unfortunately, treating the probabilistic choice both demonically (when proving the partial correctness) and angelically (when proving total correctness) might result in duplicated proof obligations elsewhere. Consider the following example, where an occurrence of $\oplus$ is followed by an operation with a precondition:

$$(S \ \oplus \ T) \ \ ; \ \ (P \mid U) \, .$$

While proving the preservation of the invariant I, we treat $\oplus$ as $[\!]$, i.e. we have to prove that both S and T establish P. However, the proof of the decrease of the variant must be handled separately, because of the angelic interpretation of $\oplus$; and in this case we find that we must prove that *either* S *or* T establishes P.

This repetition of $[S]\, P$ and $[T]\, P$ is clearly not a problem in theory, but it is certainly inconvenient in practice if the proofs require manual assistance — since it will have to be given twice. A possible solution is to rely more heavily on the theory of probabilistic loops [44], where we find that only partial correctness is required for preservation of the invariant by the loop body: partial correctness applied to preconditions allows them simply to be discarded.

For practical purpose, we can discard all preconditions of a loop body S before generating the obligations for total correctness. Let $[\![S]\!]$ be the version of $[\![S]\!]$ in which all of the preconditions are discarded. We extend Thm. 2 as follows:

Theorem 3 *Suppose we have a loop (denoted by "loop")*

WHILE G ***DO***
  S
INVARIANT I
VARIANT V
BOUND U
END ,

and we want to estimate $[\!| \mathit{loop} |\!]\, Q$*, where* Q *is a predicate. The following rules are used to generate the proof obligations for this loop.*

If

- *the invariant I is maintained during the execution of the loop (in which probabilistic choice is interpreted demonically), i.e.*

$$G \wedge I \Rightarrow [\![S]\!] I \ , \ and$$

- *on termination, the post-condition Q is established, i.e.*

$$\neg G \wedge I \Rightarrow Q \ , \ and$$

- *the variant V is natural number, i.e.*

$$G \wedge I \Rightarrow V \in \mathbb{N} \ , \ and$$

- *the variant V is bounded above by U, i.e.*

$$G \wedge I \Rightarrow V \leq U \ , \ and$$

- *the upper bound U is unchanged during the loop (in which probabilistic choice is interpreted demonically), i.e.*

$$\forall N . (G \wedge I \wedge (U = N) \Rightarrow [\![S]\!] (U = N)) \ , \ and$$

- *the variant V has some chance of decreasing (in which probabilistic choice is interpreted angelically), i.e.*

$$\forall N . (G \wedge I \wedge (V = N) \Rightarrow [\!\![S]\!\!] (V < N)) \ ,$$

then $I \Rightarrow [loop]\ Q$, i.e. we can prove I instead of $[loop]\ Q$.

The condition that the body of the loop (i.e. S) is definite is guaranteed by the use of the *abstract probabilistic choice* (which ensures that a proper probability will be used in the implementation of this choice) and the restriction of the *B-Method* that nested loops are not allowed.

This results in modifying the rules and theory for generating proof obligations accordingly.

3.4 Modifying the *B-Toolkit* for *qB*

3.4.1 System Library for Abstract Probabilistic Choice

We introduce the probability-one correctness into *B* and the result is called *B with probability-one termination* (*qB*). We keep in mind that with *qB* we want to have a simple solution for a set of problems with probability-one termination without many changes to the original *B-Method* and the supporting *B-Toolkit*. Even though the introduction of *ACHOICE* gives us a simple way of specifying probabilistic programs and reasoning about them formally, the result will not have much practical use if we cannot implement them.

The abstract probabilistic choice $\oplus$ is not actually a program construct (i.e code). It must be "refined" to or "implemented" by a proper probabilistic choice substitution. In fact, an abstract probabilistic choice cannot be refined to any deterministic or non-deterministic substitution. It must be implemented by another abstract probabilistic choice $\oplus$ or a probabilistic choice substitution ${}_p\oplus$ with a "proper" p, i.e. be bounded away from 0 and 1 (see Sec. 3.2.5). The question now is how this implementation can be done.

In fact, we say that the abstract probabilistic choice $\oplus$ is interpreted demonically everywhere (i.e. the same as the non-deterministic choice $[]$) except inside a loop. This interpretation makes the refinement of the $\oplus$ clause much harder. Without much improvement, if we simply treat $\oplus$ the same as $[]$, the proof obligation will allow refinement from $\oplus$ to $[]$, which we know to be unsound. To stop this happening, there must be a possible way to differentiate the abstract probabilistic choice $\oplus$ and the non-deterministic choice $[]$. Since $\oplus$ can refine $[]$, in most situations we still want $\oplus$ to behave as $[]$.

If we want to refine $\oplus$ during the development, then there must be some way to distinguish between $\oplus$ and $[]$, since the former is the refinement of the latter but not the other way round. Moreover, if the program contains both $\oplus$ and $[]$ then we have to record the "positions" of both $\oplus$ and $[]$. This makes the proof obligation generation impossible to implement, and takes away the simplicity that we propose by introducing $\oplus$ into *B*.

On the other hand, having abstract probabilistic choice $\oplus$ in a specification or refinement construct gains nothing when compared with using $[]$ (they are both

MACHINE *Rename_AbstractChoice* (*numerator* , *denominator*)
CONSTRAINTS
$\neg$ (*numerator* $= 0$) $\wedge$ $\neg$ (*denominator* $= 0$) $\wedge$
numerator $<$ *denominator*
SEES *Bool_TYPE*
OPERATIONS
bb $\longleftarrow$ **Rename_AbstractChoice** $\widehat{=}$
ACHOICE *bb* $:=$ *TRUE*
OR *bb* $:=$ *FALSE*
END
END

Figure 3.4: Rename_AbstractChoice system library machine

interpreted as $[\!]$). However, in implementation construct, we cannot have $\oplus$ (directly) since it is not code. This suggests the proposal of having $\oplus$ as a system library and the implementation from $\oplus$ to ${}_p\oplus$ is done during the code generation phase, i.e., we have $\oplus$ as the specification of ${}_p\oplus$. Then we can use $\oplus$ for reasoning about probabilistic-one termination, whereas the implementation to ${}_p\oplus$ is handled automatically.

A library machine, namely Rename_AbstractChoice, is created and developers will use this particular machine instead of using the *ACHOICE* clause directly. Because it is a system library of the *B-Toolkit*, the developer does not need to worry about the implementation of the Rename_AbstractChoice machine. The specification has one operation to model the effect of an abstract probabilistic choice. The specification Rename_AbstractChoice can be seen in Fig. 3.4. In this machine, the concrete probability p is specified as a rational between $numerator$ and $denominator$ which guarantees that p is "proper", i.e. bounded away from 0 and 1. This is expressed as the constraints about $numerator$ and $denominator$.

When using the *ACHOICE* clause, the developer can use machine composition clauses such as, SEES, IMPORTS to embed a copy of Rename_AbstractChoice machine. In *B* implementation, only when a machine is imported are its parameters initialised and an instance of its machine created. For Rename_AbstractChoice machine, this can be delayed until the final creation of the interface, which allows

IMPLEMENTATION *Rename_AbstractChoiceI*
REFINES *Rename_AbstractChoice*
SEES *Bool_TYPE*, *Real_TYPE*
OPERATIONS
bb ⟵ **Rename_AbstractChoice** ≙
PCHOICE numerator // denominator **OF**
bb := *TRUE*
OR
bb := *FALSE*
END
END

Figure 3.5: Implementation of Rename_AbstractChoice

the reasoning about the correctness of programs to stay in the Boolean context.

3.4.2 Syntactic and Other Changes

The introduction of the library machine Rename_AbstractChoice avoids the extensive changes in the *Analyser* and *TypeChecker*. The *Analyser* only needs to check that the *ACHOICE* clause is not allowed to be used directly. Since for every machine, it needs the *ASCII* version for its *pGSL* syntax, a new clause is added that is equivalent to the *ACHOICE* clause:

`S <--> T` is equivalent to $S \oplus T$.

This syntax is important to the proof-obligation generator. Since we want to generate the obligations for proving probability-one termination properties of programs, the proof-obligation generator for refinement is changed accordingly to incorporate the rules stated in the Sec. 3.3. This allows the obligations to be generated as before for standard programs. If the program contains loops that have *ACHOICE* clauses (or rather using the Rename_AbstractChoice operation from the library machine with the same name), the obligations are generated to prove the almost-certain termination properties.

The implementation of Rename_AbstractChoice machine can be written in *probabilistic Abstract Machine Notation* (*pAMN*) as in Fig. 3.5. The abstract prob-

```
#include "Rename_AbstractChoice.h"
#include "stdlib.h"
#include "Bool_TYPE.h"

void INI_Rename_AbstractChoice() {;}

void Rename_AbstractChoice(_bb)
int *_bb;
{
  if (
    (float)random() / (float)RAND_MAX <=
    (float)Rename_AbstractChoiceP1 / (float)Rename_AbstractChoiceP2
  ) {
    *_bb = TRUE;
  }
  else {
    *_bb = FALSE;
  }
}
```

Figure 3.6: Implementation of Rename_AbstractChoice

abilistic choice $\oplus$ is now implemented by a (concrete) probabilistic choice ${}_p\oplus$, where p is the fraction between the *numerator* and *denominator* parameters of the machine [7]. The constraint of the specification guarantees that p will be proper, and hence the refinement condition for $\oplus$ is met. The translation of the implementation of the Rename_AbstractChoice machine in *C* is done by using the random facility as shown in Fig. 3.6.

Also, since we are using *real* numbers or rather a subset of real numbers represented by fractions between natural numbers, a machine is introduced as a system library machine, namely Real_TYPE (see Fig. 3.7 on the facing page). The implementation for $frac(a, b)$ will be translated to *C* code as

$$(float)\ a\ /\ (float)\ b\ .$$

This requires a slight modification to the code-generation phase of the *B-Toolkit*.

[7] The symbol "//" is for the real division operator.

MACHINE *Real_TYPE*
SETS *REAL*
CONSTANTS *frac*
PROPERTIES $frac \in \mathbb{N} \times \mathbb{N}_1 \longrightarrow REAL$
DEFINITIONS $a \,//\, b \;\widehat{=}\; frac\,(\,a\,,\,b\,)$
END

Figure 3.7: Real_TYPE system library machine for *qB*

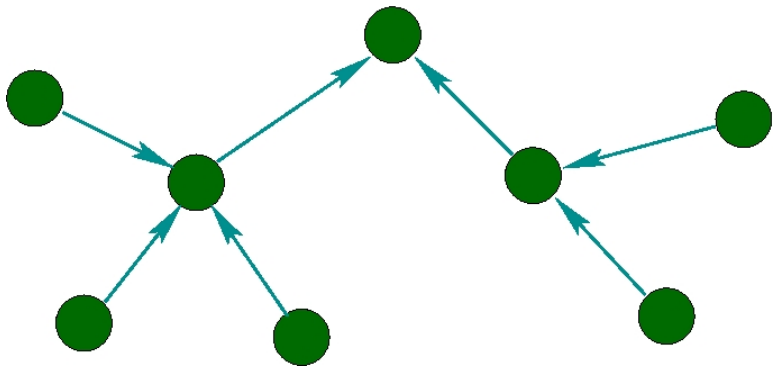

Figure 3.8: Example of message sending

3.5 Applications of *qB*

In this section, we consider two case studies based on the development of *qB*, both with probability-one termination. The first example is the Root Contention problem in the FireWire protocol [27, 28]. The second example is Rabin's Choice Coordination algorithm [57].

3.5.1 Root Contention of the FireWire Protocol

The first example for termination with probability-one is the contention resolution, part of the FireWire protocol [27, 28].

Overview of the Algorithm

We have a set of devices. The devices are connected in an acyclic network, where each device is a node in this network. The connections between nodes in the network are bidirectional. The algorithm provides a symmetric, distributed solution

Figure 3.9: Contention

to find a node that will be a leader of the network in a *finite* amount of time. All devices run the same algorithm to find the leader of the network. The algorithm is described below.

Any node with only one connection can send the message "you be the leader" via that channel to the neighbouring node. Also, any node that already received the message "you be the leader" in all its connections except one, can send the message "you be the leader" via that last remaining channel. This choice of which node sends the message is non-deterministic. Finally there will be one node that received the message "you be the leader" from all its connections and that will become the leader of the network. An example can be seen in Fig. 3.8 on the previous page.

The only problem that can happen is a livelock situation, when there are two nodes left in the network that have not sent the message "you be the leader". If those two nodes connect with each other by one channel and both try to send the message via that channel to tell the other node to be the leader of the network, then livelock can occur. The nodes recognise the problem by receiving the message "you be the leader" from the other node to which it has just sent a message. This situation can be seen in Fig. 3.9.

Fortunately, there exists a probabilistic way to resolve the situation within a finite time (which can be proved). The algorithm can be described as follows. Each node independently chooses with the same non-zero probability, either to send the message after a "short" time or after a "long" time (the assumption for the "long" time being that it is long enough for the message to be transferred from one node to another). Eventually, it is almost certain that one of them will choose to send the message in a "short" time while the other has chosen to send the message in a "long" time. The message that has been sent in a "short" time will then arrive before the other has been sent (according to the assumption). An example for solving contention can be seen in Fig. 3.10 on the next page, where one process sends a "short" message and the other sends a "long" message.

Figure 3.10: Probabilistic solution for contention problem

xx , yy ⟵ **Resolve** $\widehat{=}$

CHOICE

$xx := short\_signal \parallel yy := long\_signal$

OR

$xx := long\_signal \parallel yy := short\_signal$

END

Figure 3.11: Specification Resolve operation

Specification: Non-deterministic Resolution

We are not going to model the full FireWire protocol, but only the part related to the contention. When contention happens, two final nodes in the network try to reconfigure and find the leader for the network. The abstraction of the algorithm can be seen as one of the nodes becomes the leader non-deterministically. The symmetry of the system is broken when one of the node waits for a long time and the other waits for a short time. The Resolve operations have two outputs and the effect of the operation is a non-deterministic assignment of *short_signal* and *long_signal* to the outputs *xx* and *yy*, so that the outputs are different. This specification can be seen in Fig. 3.11. The full machine is in Appendix A.1.

Implementation: Resolving the Contention

The implementation sees a copy of the standard library machine AbstractChoice to gain access to an operation that represents abstract probabilistic choice (Fig. 3.12 on the next page). Two local variables *tempx* and *tempy* will represent the status of the nodes during the process of breaking the contention. Initially, they are given the same values (i.e. there is a contention).

Each node will either choose to send the message in a short time or to wait for a long time (to have status *short_signal* or *long_signal* respectively) with some non-zero probability. This maintains the fairness between choosing two different

```
xx , yy ⟵ Resolve ≙
  VAR tempx , tempy , bb IN
    tempx := short_signal ; tempy := short_signal ;
    WHILE tempx = tempy DO
      bb ⟵ Firewire_AbstractChoice ;
      IF bb = TRUE THEN tempx := short_signal
      ELSE tempx := long_signal END ;
      bb ⟵ Firewire_AbstractChoice ;
      IF bb = TRUE THEN tempy := short_signal
      ELSE tempy := long_signal END
    BOUND 1
    VARIANT prob ( tempx = tempy )
    INVARIANT tempx ∈ STATUS ∧ tempy ∈ STATUS
    END ;
    xx := tempx ; yy := tempy
  END
```

Figure 3.12: Implementation Resolve operation

branches. A process cannot choose to send the message in a short time forever, and vice versa. The contention is resolved (the loop is terminated) when two nodes choose different values. Here, the values of *tempx* and *tempy* are changed according to the result of the corresponding abstract probabilistic choice substitution.

The variant of the loop is the embedded predicate $\langle tempx = tempy \rangle$ [8] — denoted by $prob(tempx = tempy)$ — which is the probability that *tempx* will be the same as *tempy*. Clearly this variant is bounded above by 1 (by definition of embedded predicates). The guard of the loop guarantees that the variant has value 1 before every iteration of the loop. And there is always a non-zero chance for the variant to decrease to 0, i.e. that *tempx* and *tempy* are different.

The invariant is simply a type predicate for the local variable *tempx* and *tempy*. On termination, the guard will guarantee that $tempx \neq tempy$, and also that the outputs *xx* and *yy* are different, which satisfies our specification.

In this implementation, the abstract probabilistic choice comes from a seen

[8] Recall that the embedded predicate $\langle P \rangle$ has the value 1 if the predicate *P* is *true* and 0 otherwise.

Resolve.5

$cst\ (\ FirewireResolveI_{\ 1}\ ) \wedge ctx\ (\ FirewireResolveI_{\ 1}\ ) \wedge$

$inv\ (\ FirewireResolveI_{\ 1}\ ) \wedge asn\ (\ FirewireResolveI_{\ 1}\ ) \wedge$

$pre\ (\ Resolve\ )$

$\Rightarrow$

$tempx \in STATUS \wedge$

$tempy \in STATUS \wedge$

$tempx = tempy$

$\Rightarrow$

$prob\ (\ true\ ) < prob\ (\ true\ )$

$\vee$

$prob\ (\ false\ ) < prob\ (\ true\ )$

$\vee$

$prob\ (\ false\ ) < prob\ (\ true\ )$

$\vee$

$prob\ (\ true\ ) < prob\ (\ true\ )$

Figure 3.13: *Resolve.5* obligation

Construct	Obs	Automatic	Manual	Remained
FirewireResolve.mch	0	0	0	0
FirewireResolveI.imp	7	7 (in 2 levels)	0	0

Table 3.1: Proof obligations summary for Root contention problem

copy of the system library machine, namely Firewire_AbstractChoice. The actual probability that is used will be instantiated when the Firewire_AbstractChoice machine is included in the development. In this case, it is introduced into the interface construct, i.e. it is instantiated when the actual interface is generated for executable code. This does not effect the correctness of the implementation, since as long as the input probability satisfies the constraint of Firewire_AbstractChoice machine (hence guaranteeing that it is proper), the program will terminate with probability one. The full implementation is in Appendix A.2.

Proof Obligations

Tab. 3.1 is the summary for proof obligations generated and discharged with the root contention algorithm using the modified *B-Toolkit*.

Some proof rules need to be added to the rules base in order to discharge the obligations for FirewireResolve! implementation. These are to assert facts about embedded predicates, such as $prob(true) = 1$ and $prob(false) = 0$. The obligation *Resolve.5* for proving the termination with probability one is shown in Fig. 3.13 on the preceding page. The disjunctions show that the abstract probabilistic choice is interpreted angelically in this case. It means that the variant has some chance (not necessary always) of decreasing.

So we have proved that the correct implementation for the contention and the implementation does in fact terminate with probability one.

3.5.2 Rabin's Choice Coordination

The second example of applying probability one termination is Rabin's Choice Coordination algorithm [57].

Overview of the Algorithm

The following is the summary of the problem and algorithm by Rabin in [57].

The Choice Coordination is the problem in which we have n different processes (namely $P_1, P_2, \ldots, P_n$) with k possible alternatives $(A_1, A_2, \ldots, A_k)$ for the computations of these processes. Each process must choose only one of these alternatives to execute. It does not matter which alternative to be chosen, but all processes must choose the same alternative. It is assumed that each process has its own system name, and for each alternative A_i where $1 \leq i \leq k$, there is a shared variable v_i for $1 \leq i \leq k$ that can be accessed and modified by every process in one indivisible step that cannot be interrupted by any other process. The algorithm tries to break the inherent symmetry (where we are assuming that every process is running the same algorithm).

We consider the slightly simplified version of the algorithm from Morgan [49] which is described in terms of tourists. There is a group of tourists who try to agree on going to either the church (assumed to be on the left) or the museum (assumed to be on the right). In this case, the number of alternatives is 2. Each tourist carries

a notepad that he/she uses to write a number on the notepad. There are two notice boards outside the two places that can be used to write on. Usually, a number is written on the notice board, but alternatively, a word "Here" can be written on it as well. Originally, the number 0 appears on all of the tourists' notepads and the notice boards outside the two places.

Each tourist carries the notepad and chooses an initial place to go to, and then alternates between the two places according to the following algorithm:

- If the notice board outside the place has the word "Here" then he goes inside this place.
- Otherwise, the notice board will display a number L. By comparing this number with the number K on his notepad, the tourist can do one of the followings:
  1. If $K > L$, he writes "Here" on the board then goes inside (deleting the number L on the board).
  2. If $K = L$, he then chooses a number $K' = K + 2$, then tosses a coin. If the coin turns up heads then he keeps K' the same, otherwise he sets K' to its "conjugate"[9] . He then writes K' on the board and on his notepad before going to the other place.
  3. If $K < L$, he replaces K with L on his notepad, and then goes to the other place.

The algorithm is guaranteed to terminate with probability-one with all the tourists inside either the church or the museum. We will investigate the correctness of this algorithm by using *B* machines, refinements, and implementations. We start with a very simple abstract specification and perform step by step refinements of this specification.

Specification: Non-deterministic Destination

The specification contains no variables and has only one operation namely Decide. The inputs $lout$ and $rout$ of the operation represent the numbers of people originally outside the two places: the church and the museum. The outputs

[9] conjugate of a number n is defined to be $n + 1$ if n is even and $n - 1$ if n is odd.

lin , rin $\longleftarrow$ **Decide** ($lout$, $rout$) $\widehat{=}$

PRE

$lout \in \mathbb{N} \wedge rout \in \mathbb{N} \wedge$

$lout + rout \leq maxtotal$

THEN

CHOICE

$lin := lout + rout \parallel rin := 0$

OR

$rin := lout + rout \parallel lin := 0$

END

END

Figure 3.14: Specification of Decide operation

lin and rin of the operation represent the numbers of people inside the places afterwards. The behaviour of the operation is non-deterministic since the algorithm does not specify where all the tourists will go. There are two possibilities: either all the tourists will end up inside the left place (the church), in this case $lin := lout + rout$ and $rin := 0$; or they will all go inside the right place where $lin := 0$ and $rin := lout + rout$.

The precondition in this case ensures that the total number of tourists does not exceed a maximum number, which is specified as a parameter of the machine. This is purely for the purpose of implementation in the later stage of the development when importing the system libraries which have constraints on the their parameters. The specification is shown in Fig. 3.14.

Refinement: Using Bags

In this step, we refine the above simple specification using the concepts of bags. First of all, we introduce the specification of a "finite bag". "Bag" is a mathematical construct containing multiple elements. Unlike "Set", that can only contain distinct elements, "bags" can contain multiple occurrences of the same element. In the specification, we model a finite bag of natural numbers as a function that maps indexes to elements in the bag. The domain of the function is a finite set of natural number to ensure that the bag is finite. Two indexes can map to the same number

lin , rin $\longleftarrow$ **Decide** ($lout$, $rout$) $\widehat{=}$

BEGIN

ANY $flinbag$, $frinbag$, $floutbag$, $froutbag$

WHERE

$flinbag \in Bag \wedge frinbag \in Bag \wedge floutbag \in Bag \wedge froutbag \in Bag \wedge$

$\mathsf{dom}\ (\ flinbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge \mathsf{dom}\ (\ frinbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge$

$\mathsf{dom}\ (\ floutbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge \mathsf{dom}\ (\ froutbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge$

$floutbag = \{\} \wedge froutbag = \{\} \wedge$

$(\ bagSize\ (\ flinbag\ ) = 0 \vee bagSize\ (\ frinbag\ ) = 0\ ) \wedge$

$bagSize\ (\ flinbag\ ) + bagSize\ (\ frinbag\ ) = lout + rout$

THEN

$lin\ .\ SetToBag\ (\ flinbag\ )\ \|\ rin\ .\ SetToBag\ (\ frinbag\ )\ \|$

$lout\ .\ SetToBag\ (\ floutbag\ )\ \|\ rout\ .\ SetToBag\ (\ froutbag\ )$

END ;

$lin \longleftarrow lin\ .\ Size\ \|\ rin \longleftarrow rin\ .\ Size$

END

Figure 3.15: First refinement of Decide operation

to represent the capability of having multiple elements in a bag.

The specification has operations to set the contents of the bag (SetToBag operation), take out one element from the bag (Takelem operation), add one element to the bag (Addelem operation), non-deterministically get one element from the bag (Anyelem operation) and a query operation (Size) to get the size of the bag. The details of the FBag machine are given in Appendix B.1.

The context machine FBag_ctx contains context information (definitions) related to finite bags. It include the definitions of *Bag* and some other mathematical constructs related to bags, such as, the size of a bag and the maximum element in the bag. This machine also can be seen in Appendix B.2.

Using the idea of finite bags, we can represent the tourists in our algorithm. Our tourists will be in one of four places: inside or outside, or the left or right places (the church or the museum). Each tourist carries a notepad with a number on it, and this will be the only characteristic of the tourist. Hence we can use four finite bags to represent four groups of tourists. With the mechanism of renaming in

the INCLUDES clause (see Sec. 2.2), we can have four instances (namely, *lin*, *rin*, *lout*, *rout* for inside-left, inside-right, outside-left and outside-right, respectively) of the finite bag specified above.

The operation Decide is now described in terms of these four bags. We still use a non-deterministic clause (*ANY*) to specify the final values of the four bags. The two bags that represent tourists outside two places (*lout* and *rout*) will be empty, whereas only one of the bags representing tourists inside will be empty. The total number of tourists is unchanged, i.e all the tourists will be gathered in the place represented by the non-empty bag. This refinement can be seen in the Fig. 3.15 on the preceding page.

Implementation: Using Abstract Probabilistic Choice

In the implementation of the Rabin's algorithm, an instance of the AbstractChoice system machine, namely tourist_AbstractChoice is seen. This will allow access to an abstract probabilistic choice substitution in the tourist_AbstractChoice operation. The implementation of the Decide operation (Fig. 3.16 on the next page) starts with the InitState operation from the RabinState machine. This step corresponds to a different set of people initially going to different places (where outside the church there are *lout* people, and outside the museum there are *rout* people).

The main body of the implementation is a *WHILE* loop. The action of tourists moving from one place to another continues until all of them end up in the same place, i.e. the number of people outside the two places is zero ($sizelout = 0 \wedge sizerout = 0$). While there are still some people outside the destinations, a tourist updates his pad according to the algorithm. And this step continues after checking whether there are some people outside the two places. Here, we use the definition of "conjugate" numbers when specifying the updating of the tourists' notepads.

When the loop terminates, the sizes of the bag *lin* and *rin* are assigned to the outputs accordingly. The invariant of the loop guarantees that the temporary variables *sizelout* and *sizerout* hold the number of people outside the two places, and the total number of tourists is unchanged during the execution of the algorithm. Another invariant is needed (and is explained in [49]). It is that there are no tourists outside the right place that have the same number on their notepad as the conjugate of the number on the board at the left place and vice versa. This is expressed as

$$\neg\ (\ \mathit{Conjugate}\ (\ \mathit{LLval}\ )\ \in \mathsf{ran}\ (\ \mathit{routbag}\ ))\ .$$

lin , *rin* ⟵ **Decide** (*lout* , *rout*) ≙

VAR *sizelout* , *sizerout* , *bb* , *sizelin* , *sizerin* , *LLval* , *RRval* , *rr* , *ll* **IN**

InitState (*lout* , *rout*) **;** *LLval* ⟵ *LLVal* **;** *RRval* ⟵ *RRVal* **;**

sizelout ⟵ *loutSize* **;** *sizerout* ⟵ *routSize* **;** *sizelin* ⟵ *linSize* **;** *sizerin* ⟵ *rinSize* **;**

WHILE *sizelout* ≠ *0* ∨ *sizerout* ≠ *0* **DO**

IF *sizelout* ≠ *0* **THEN** *ll* ⟵ *loutAnyelem* **;**

IF *sizelin* = *0* ∧ *ll* < *LLval* **THEN** *MoveToRight* (*ll* , *LLval*)

ELSIF *sizelin* ≠ *0* ∨ *ll* > *LLval* **THEN** *MoveInLeft* (*ll*)

ELSE *LLval* := *LLval* + *2* **;** *bb* ⟵ *tourist_AbstractChoice* **;**

IF *bb* = *TRUE* **THEN** *MoveToRight* (*ll* , *LLval*)

ELSE *LLval* ⟵ *ConjugateCal* (*LLval*) **;** *MoveToRight* (*ll* , *LLval*) **END**

END

ELSE *rr* ⟵ *routAnyelem* **;**

IF *sizerin* = *0* ∧ *rr* < *RRval* **THEN** *MoveToLeft* (*rr* , *RRval*)

ELSIF *sizerin* ≠ *0* ∨ *rr* > *RRval* **THEN** *MoveInRight* (*rr*)

ELSE *RRval* := *RRval* + *2* **;** *bb* ⟵ *tourist_AbstractChoice* **;**

IF *bb* = *TRUE* **THEN** *MoveToLeft* (*rr* , *RRval*)

ELSE *RRval* ⟵ *ConjugateCal* (*RRval*) **;** *MoveToLeft* (*rr* , *RRval*) **END**

END

END ;

sizelout ⟵ *loutSize* **;** *sizerout* ⟵ *routSize* **;** *sizelin* ⟵ *linSize* **;** *sizerin* ⟵ *rinSize*

BOUND *lexicographic* (*2* , *0*)

VARIANT

lexicographic (*rEqual* (*LLval* , *RRval*) ,

3 × (*bagSize* (*linbag*) + *bagSize* (*rinbag*)) +

(*bagGreat* (*loutbag* , *LLval*) + *bagGreat* (*routbag* , *LLval*)) +

(*bagGreat* (*loutbag* , *RRval*) + *bagGreat* (*routbag* , *RRval*)))

INVARIANT

LL = *LLval* ∧ *RR* = *RRval* ∧ *LLval* ↦ *RRval* ∈ dom (*rEqual*) ∧ *total* = *lout* + *rout*

¬ (*Conjugate* (*LLval*) ∈ ran (*routbag*)) ∧

¬ (*Conjugate* (*RRval*) ∈ ran (*loutbag*)) ∧

bagSize (*loutbag*) = *sizelout* ∧ *bagSize* (*routbag*) = *sizerout* ∧

bagSize (*linbag*) = *sizelin* ∧ *bagSize* (*rinbag*) = *sizerin* ∧

END ;

lin ⟵ *linSize* **;** *rin* ⟵ *rinSize*

END

Figure 3.16: Implementation of Decide operation

The variant is defined to be a natural number corresponding to the lexicographical variant defined in [49]. Since the algorithm is to terminate with probability-one, we have to specify the upper bound of the variant (see Sec. 3.3). In this case the upper bound is defined using the *BOUND* clause, and a proof obligation will be generated to prove that the variant is "always" bounded above by this constant (i.e. $lexicographic(2, 0)$). The basis properties of $lexicographic$ are as follows:

$$lexicographic(a, b) > lexicographic(c, d) \quad \text{iff} \quad a > c$$

and

$$lexicographic(a, b) > lexicographic(a, c) \quad \text{iff} \quad b < c \ ,$$

for a, b, c, d are natural numbers. In the variant, we use the definition $rEqual$ which can be seen in Fig. 3.17 on the facing page. The term $rEqual$ is defined according to the definition of conjugate numbers.

The main part of the algorithm (how the tourist updates his/her notepad) is represented by two branches of the *IF* clause. Here, the scheduler gives the favour to the case where there are some people outside the left place. (In the ideal situation, it should be able to have a non-deterministic choice between the two cases, or even better, a probabilistic choice). The implementation includes the machine RabinState to model the sets of people in the different places. The program is symmetric in terms of left and right (except for the choice that explained above). For example, if there are some people outside the left place, any person from the left place can have his turn, i.e.

$$ll \longleftarrow loutAnyelem \ .$$

Then depending on the relationship between the number on his pad (*ll*) and the number on the board (*LLval*), and whether or not there are already some people inside the left place, the tourist responds accordingly by calling operations such as MoveToRight or MoveInLeft. When there are no people inside the left place, and the number on his pad equals to the number on the board, i.e.

$$ll = LLval \ ,$$

he increases his number by 2. The variable *bb* represents the output of the operation tourist_AbstractChoice (the outcome of the tossing a coin). Depending on the

PROPERTIES

$rEqual \in \mathbb{N} \times \mathbb{N} \pfun \mathbb{N} \land$

$\forall\, (\, mm \,,\, nn \,) \,.\, (\, mm \in \mathbb{N} \land nn \in \mathbb{N} \land mm = nn \Rightarrow rEqual\, (\, mm \,,\, nn \,) = 2 \,) \land$

$\forall\, (\, mm \,,\, nn \,) \,.\, (\, mm \in \mathbb{N} \land nn \in \mathbb{N} \land$
$(\, mm = Conjugate\, (\, nn \,) \lor nn = Conjugate\, (\, mm \,) \,) \Rightarrow$
$rEqual\, (\, mm \,,\, nn \,) = 0 \,) \land$

$\forall\, (\, mm \,,\, nn \,) \,.\, (\, mm \in \mathbb{N} \land nn \in \mathbb{N} \land$
$(\, minConjugate\, (\, mm \,) = minConjugate\, (\, nn \,) + 2 \lor$
$minConjugate\, (\, nn \,) = minConjugate\, (\, mm \,) + 2 \,) \Rightarrow$
$rEqual\, (\, mm \,,\, nn \,) = 1 \,)$

$Conjugate \in \mathbb{N} \fun \mathbb{N} \land$
$\forall\, nn \,.\, (\, nn \in \mathbb{N} \land nn \mod 2 = 0 \Rightarrow Conjugate\, (\, nn \,) = nn + 1 \,) \land$
$\forall\, nn \,.\, (\, nn \in \mathbb{N} \land nn \mod 2 = 1 \Rightarrow Conjugate\, (\, nn \,) = nn - 1 \,) \land$

$minConjugate \in \mathbb{N} \fun \mathbb{N} \land$
$\forall\, nn \,.\, (\, nn \in \mathbb{N} \land nn \mod 2 = 0 \Rightarrow minConjugate\, (\, nn \,) = nn \,) \land$
$\forall\, nn \,.\, (\, nn \in \mathbb{N} \land nn \mod 2 = 1 \Rightarrow minConjugate\, (\, nn \,) = nn - 1 \,) \land$

Figure 3.17: Definition of $rEqual$

value of *bb*, the tourist either keeps the number the same or changes to its conjugate, and then goes to the right. The algorithm behaves similarly for people on the right.

The machine RabinState consists of four instances of FBag to represent four groups of people in four different places. The machine provides operations corresponding to the the movement of tourists from place to place, where they perform different actions. For example the operation MoveInLeft in Fig. 3.18 moves a person with the number *ll* on his pad from outside to inside the left place.

The operation InitState in Fig. 3.19 on the next page sets the bags to represent the initial state of the algorithm: there are no tourists inside the church or the

MoveInLeft (ll) $\widehat{=}$

PRE

$ll \in \mathsf{ran}\ (\ loutbag\ ) \wedge (\ bagSize\ (\ linbag\ ) \neq 0 \vee ll > LL\ )$

THEN

$lout\ .\ Takelem\ (\ ll\ )\ \|$

$lin\ .\ Addelem\ (\ ll\ )$

END

Figure 3.18: MoveInLeft operation

InitState ($lout$, $rout$) $\widehat{=}$

PRE $lout \in \mathbb{N} \wedge rout \in \mathbb{N} \wedge lout + rout \leq maxtotal$ **THEN**

$lin\ .\ SetToBag\ (\ \{\}\ )\ \|\ rin\ .\ SetToBag\ (\ \{\}\ )\ \|$

$lout\ .\ SetToBag\ (\ (\ 1\ ..\ lout\ ) \times \{\ 0\ \}\ )\ \|$

$rout\ .\ SetToBag\ (\ (\ 1\ ..\ rout\ ) \times \{\ 0\ \}\ )\ \|$

$total := lout + rout\ \|$

$LL := 0\ \|\ RR := 0$

END ;

Figure 3.19: InitState operation

museum, outside the church (on the left) there are *lout* tourists, and outside the museum (on the right) there are *rout* tourists.

Similar to the previous example, the actual probability used in this case is instantiated in the interface construct.

All the machines, refinements and implementations mentioned here are given in Appendix B.

Proof Obligations

The table below is a summary of the proof obligations generated and discharged for Rabin's Choice Coordination algorithm using the modified *B-Toolkit*.

For those obligations that have been discharged manually, all of them are for total correctness, i.e. to prove the angelic decrease of the variant. (The last two obligations are very difficult to prove). The partial correctness of the implementation is carried by the invariants from RabinChoice and RabinState. Some proof

Construct	Obs	Automatic	Manual	Remained
Rabin.mch	1	1	0	0
RabinR.ref	1	1	0	0
RabinRI.imp	103	83 (in 2 levels)	20 (in 2 levels)	0
RabinState.mch	46	39 (in 2 levels)	7 (in 1 level)	0

Table 3.2: Proof obligations summary for Rabin's Coordination

rules are added to the rule base in order to discharge the obligations. These proof rules are related to the concept of bags, and to facts about some mathematical constructs that are used in the development.

We have developed and proved the correctness of programs implement Rabin's Choice Coordination algorithm. We developed the algorithm in layers, in order to separate the complexity of the system. Using finite bags, we are able to abstractly represent the tourists who are identified by the number on their notepads. Furthermore, we are able to prove the probability-one termination of the algorithm with the modified *B-Toolkit*.

3.6 Conclusions and Related Work

In this chapter, we presented the idea of probability-one termination and introduced abstract choice substitution ($\oplus$). We discussed how abstract choice can be implemented into *B* and supported by the modified *B-Toolkit*. The two examples that we used in this chapter has been developed and proved completely using the modified *B-Toolkit*.

Here, we did not focus on the quantitative side of a probabilistic program (e.g. expected time for termination, etc.). To calculate such values, we would need to have the full *pGSL* in order to reason numerically. We chose to separate the proof of termination by using a simple logic, which resulted in a simple extension of *B*, but it is still expressive enough to cover the important issues of termination. This led to the practical result of having a tool to support the development of programs (including generating proof obligations and proving such obligations). The mathematical foundations can also be found in earlier work by Morgan and McIver for temporal logic [47, 48]. The work introduced in this chapter is based on their theoretical work.

A similar approach has been pursued by Rao [58, 59] using *UNITY* programs. It includes the definition of a UNITY-*style fairness* assumption.

Similarly Pnueli [54, 22] considered the termination of probabilistic concurrent programs and introduced the concept of *extreme fairness*. This fairness is later on extended to α-fairness by Zuck [69] for past-time temporal logic. The concept of probability-one correctness has also been used in other work, in which it has been called *P-valid*. The concept of γ-fairness is close to what we have called the angelic interpretation of probabilistic choice: if a probabilistic choice is invoked infinitely many times from a given state then each branch of the choice should be taken from that state [70].

For the FireWire protocol, there is some other work by Fidge and Shankland, and by Abrial using probabilistic- and standard predicate transformers respectively. This is summarised by Stoelinga in [64]. The example also introduces for mechanical verification methods by Stoelinga, as in [62, 65].

The question of "how long?" was asked by Fidge and Shankland, and in order to investigate the expected value of time, we would need to use numeric logic instead of simple probability-one reasoning. In their work [17], Fidge and Shankland use the *probabilistic Guarded Command Language* (*pGCL*) which is a probabilistic extension of *GCL* (the original Dijkstra's version of *GSL*).

Moreover, Abrial's development of the FireWire protocol [5] was done in *Event-B* [3, 6] which is the successor of the (now called) original *B*. In this development, the contention is resolved by some fairness assumptions which is in fact the intuition behind this work. This also leads some new concerns which could be regarded as future work.

Event-B is an event based style of development in which systems comprise of events (which can be enabled) rather than operations (which can be called). There are no explicit loops in an event system. Instead the whole system is a *WHILE* loop in which the body of the loop is essentially the non-deterministic choice of all the events. In *Event-B*, an abstract system can be refined by a concrete system with more events, but there must be a corresponding event to each event in the abstract system as before. The set of new events generates a different set of obligations according to the following rules:

- each new event refines skip event; and
- each new event decreases the same variant.

The second rule ensures that the new events cannot take over the system and they must eventually deadlock, in order to allow the original events to be enabled.

If we want to keep the same terminology of *Event-B*, i.e. to have naked guarded commands with a simple body as events, then the reasoning about probabilistic choice (both abstract and concrete) can exist between events. Here, we are only concerned only about the abstract choice. Then the second rule would have to be changed so that new events probabilistically decrease the same variant. The notion of $[\![\cdot]\!]$ should be useful with this new rule.

In our case studies, we only consider a small fraction of the real protocol (in the case of FireWire), and our final systems are not distributed. But we can consider these systems as faithful abstractions of the real systems.

Chapter 4

Probabilistic Machines

4.1 Numerical Reasoning

In the previous chapter, in order to cover probability-one-termination correctness, we extended *B* to *qB* by introducing the abstract probabilistic choice operator $\oplus$ (and its corresponding *AMN* clause). The overall method of reasoning for *qB* is still in Boolean logic. The set of problems that *qB* covers includes distributed systems where probability is used to break symmetry. However, *qB* cannot handle numerical reasoning such as the expected time to terminate, or the expected values of some expressions over the state.

In this chapter, we enable numerical reasoning by introducing the (concrete) probabilistic choice substitution ${}_p\oplus$, and by using the semantics of expectations to reason about our programs. Recall from Sec. 2.5 that *GSL* can be extended to *pGSL*, in which the standard Boolean values —representing certainty— are replaced by real values —representing probabilities. Hence, in principle, the *standard* machines of *B* can be extended to *probabilistic B-Method* (*pB*) machines, allowing us to implement random algorithms, or to model faulty (unreliable) operations. Moreover, we need to have a tool to support the probabilistic constructs enabling us to generate and discharge proof obligations. For practical purposes, we want to retain the Boolean reasoning as much as possible by separating the probabilistic and standard constructs. This enables proofs to be conducted in the existing Boolean logic extended by the set of rules supporting reals.

We have already presented the foundational issues on probabilistic computa-

tional models, in Sec. 2.5. One important point to notice is that the theory on which the current chapter is based [40] requires that those "expectations" (real-value functions from state) are non-negative and bounded. For simplicity, however, the examples presented in this chapter do not necessarily adhere to these constraints. Here, we only discuss the development and tool support of probabilistic machines based on *pGSL*.

This chapter extends the concept of *invariant* to probabilistic machines, based on the theory of *pGSL*, as follows: we define *probabilistic invariants*; we set out the proof obligations for maintaining such invariants; we give informal interpretations of the meaning of these invariants in practice; we develop a machine construct and give examples of how to use it; and we highlight possible pitfalls, and possible approaches to correct them.

4.2 Probabilistic Invariant for Probabilistic Machines

In this section, we focus on our main topic of the chapter: the meaning of expectations when used as "invariants" for a *pB* machine. We will use the well known example of a library to illustrate our ideas here.

We begin with a standard specification and use it as a basis against which we can contrast a probabilistic version. Our aim is, first, to show how probabilistic invariants capture their probabilistic properties and, second, to highlight some of the unexpected and subtle issues that can arise.

4.2.1 A Simple Library in *B*

Consider the specification of a simple library in Fig. 4.1 on the facing page. The state of the machine contains three variables, namely $booksInLibrary$, $loansStarted$ and $loansEnded$, representing the number of books in the library, the number of book loans initiated by the library, and the number of book loans completed by the library, respectively. To keep the example simple, we ignore other functions of the library (such as user management, etc.). Initially, $booksInLibrary$ has value $totalBooks$ (a parameter of the machine). Both $loansStarted$ and $loansEnded$ are assigned 0 initially.

We have two operations that can modify the state of the machine: StartLoan,

MACHINE *StandardLibrary* (*totalBooks*)
VARIABLES
booksInLibrary , *loansStarted* , *loansEnded*
INVARIANT
$booksInLibrary \in \mathbb{N} \wedge loansStarted \in \mathbb{N} \wedge loansEnded \in \mathbb{N} \wedge$
$loansEnded \leq loansStarted \wedge$
$booksInLibrary + loansStarted - loansEnded = totalBooks$
INITIALISATION
booksInLibrary , *loansStarted* , *loansEnded* := *totalBooks* , *0* , *0*

OPERATIONS
StartLoan $\widehat{=}$
PRE $booksInLibrary > 0$ **THEN**
$booksInLibrary := booksInLibrary - 1 \;\|$
$loansStarted := loansStarted + 1$
END ;

EndLoan $\widehat{=}$
PRE $loansEnded < loansStarted$ **THEN**
$booksInLibrary := booksInLibrary + 1 \;\|$
$loansEnded := loansEnded + 1$
END

END

Figure 4.1: Standard specification of a Library

for starting a loan of a book, and EndLoan, for ending the loan of a book. The StartLoan operation has a precondition that there are books available for loan: the operation decrements the books held and increments the books loaned. The EndLoan operation is complementary in the obvious way.

The (standard) invariant of this machine (which is declared by the INVARIANT clause in Fig. 4.1) is (we ignore some other typing and trivial invariants)

$$booksInLibrary + (loansStarted - loansEnded) \quad = \quad totalBooks \,, \qquad (4.1)$$

in which the term "$loansStarted - loansEnded$" is an abstraction of the number of books that are in the on-loan database of the library. Notice that when there are books that are lost, the number of books that recorded in the database might be different from the "actual" number of books that have been borrowed. In this case however, these numbers are the same.

4.2.2 Adding Probabilistic Properties to the Library

In the Boolean world of standard *B*, the operations and invariants express certainty: books are either in the library or they are on loan; they cannot be anywhere else. In a real library, books are occasionally lost. In this section, we discuss how losing books can be modelled in *pB*.

In the first approach, we might consider adding a *Lose* operation of the form

$$Lose \quad \widehat{=} \quad booksInLibrary := booksInLibrary - 1 \ , \tag{4.2}$$

and to arrange that every so often Lose is invoked, with some probability. The problem with this is that we have no way in *B* (or in *pB* for that matter) of modelling a probabilistically invoked operation. (Later on, we discuss a similar approach for *Event-B* as future work.)

However, we can use a probabilistic choice substitution to model operations with probabilistic *effects* in *pB*, and so we take this approach. Again, for simplicity, we consider the case of losing a book when returning only. Obviously, in reality, books can be lost in other situations, including borrowing. Here, the loss of a book will be modelled by altering the EndLoan operation so that, with some probability pp, the user fails to return a book to the library; in that case, the effect of EndLoan is to consider the book as lost. The other $1-pp$ of the time, the book is successfully returned.

With the introduction of probabilistic choice substitution (as in Sec. 2.5), we can specify this behaviour within the EndLoan operation. The PCHOICE construct is the *pAMN* counterpart of the operator ${}_p\oplus$, i.e.

PCHOICE p **OF**
  S
OR
  T
END

corresponds to $S \ {}_p\oplus \ T$.

We introduce the variable $booksLost$ to keep the number of books lost. We initialise $booksLost$ to 0 —there are no books lost at the beginning. For the EndLoan operation, the chance of a book being lost is pp —where $booksInLibrary$ fails to increase; the other $1 - pp$ of the time, $booksInLibrary$ increases as normal. In the case of losing a book, $booksLost$ will increase accordingly. Hence, we replace the standard substitution $booksInLibrary := booksInLibrary + 1$ with the following:

PCHOICE *pp* **OF**

$booksLost := booksLost + 1$

OR

$booksInLibrary := booksInLibrary + 1$

END

To take into account the new variable *booksLost*, we need to modify the (standard) invariant (4.1) as follows:

$$\begin{array}{rcl} & (booksInLibrary + booksLost) & \\ + & (loansStarted - loansEnded) & = \quad totalBooks\ . \end{array} \tag{4.3}$$

The first term on the left-hand side is the number of books not in the on-loan database; the second term is the number of books that are in the on-loan database. This specification is simply modelling the effect of loss, without attempting to identify where it occurs. In practice, loss could be the consequence of a faulty (unreliable) loan or return operation. At some point, "loss" needs to be recognised and that is modelled by the assignment $booksLost := booksLost + 1$. Here, we choose to model the loss as part of the EndLoan process. The full specification of the probabilistic library can be seen in Fig. 4.2 on the following page.

4.2.3 The EXPECTATIONS Clause

Notice in Fig. 4.2 on the next page we introduce a new EXPECTATIONS clause into *pAMN* for declaring the probabilistic invariant. It gives an expression E over the program variables, denoting the (real-value) random-variable invariant, and an initial expression e which is evaluated over the program variables when the machine is initialised. We write this as

$$e \Rrightarrow E\ .$$

MACHINE *ProbabilisticLibrary* (*totalBooks*)
SEES *Real_TYPE*
CONSTANTS *pp*
PROPERTIES $pp \in REAL \wedge pp \leq real(1) \wedge real(0) \leq pp$
VARIABLES
booksInLibrary , *loansStarted* , *loansEnded* , *booksLost*
INVARIANT
$booksInLibrary \in \mathbb{N} \wedge loansStarted \in \mathbb{N} \wedge$
$loansEnded \in \mathbb{N} \wedge booksLost \in \mathbb{N} \wedge$
$loansEnded \leq loansStarted \wedge$
$booksInLibrary + booksLost + loansStarted - loansEnded = totalBooks$
EXPECTATIONS
$real(0) \Rrightarrow pp \times real(loansEnded) - real(booksLost)$
INITIALISATION
$booksInLibrary := totalBooks \parallel$
$loansStarted , loansEnded , booksLost := 0 , 0 , 0$

OPERATIONS
StartLoan $\widehat{=}$
PRE $booksInLibrary > 0$ **THEN**
$booksInLibrary := booksInLibrary - 1 \parallel$
$loansStarted := loansStarted + 1$
END ;
EndLoan $\widehat{=}$
PRE $loansEnded < loansStarted$ **THEN**
PCHOICE pp **OF**
$booksLost := booksLost + 1$
OR
$booksInLibrary := booksInLibrary + 1$
END $\parallel$
$loansEnded := loansEnded + 1$
END

END

Figure 4.2: Simple probabilistic Library

Its interpretation is that that the expected value of E, at any point, is always at least e. The value of e can depend on the context of the machine (i.e. the machine's parameters, constants, etc.), but often e will be a constant. For example, in the library, e can be an expression over the parameter $totalBooks$ and the constant pp.

4.2.4 What Do Probabilistic Invariants Guarantee?

We answer the above question by an analogy with standard invariants, which we review first.

Suppose a machine has initialisation *init* and two operations OpX and OpY. Here, for simplicity, we assume the operations have no preconditions and also omit references to the context of the machine. If we satisfy the standard proof obligations with respect to some invariant I, viz.

$$\begin{aligned} \mathsf{true} &\Rightarrow [\mathit{init}]\, I \\ I &\Rightarrow [\mathsf{OpX}]\, I \\ I &\Rightarrow [\mathsf{OpY}]\, I, \end{aligned} \tag{4.4}$$

then we are assured that

$$\mathsf{true} \Rightarrow [\mathit{init}; \mathsf{Op?}; \mathsf{Op?}; \ldots; \mathsf{Op?}]\, I \tag{4.5}$$

holds for any (finite) sequence of operations Op? each chosen from the set of operations $\{\mathsf{OpX}, \mathsf{OpY}\}$. Here, the individual choice for Op? is arbitrary (as long as the precondition of chosen operation is satisfied). The ordering of operations and the decision on when to stop are irrelevant. Operation sequences could be chosen in advance or "on-the-fly". In both cases, the maintenance of the invariant does not depend on the decision of when operations of the machine are invoked.

The fact that (4.4) assures (4.5) is an expression of the *soundness* of the (standard) invariant technique.

For soundness of the probabilistic invariant technique, clearly there must be a similar situation —that is, a probabilistic version of (4.4) and (4.5)— with the first implying the second. (Again, we omit the preconditions of the operations and the context of the machine).

MACHINE *Counter*
SEES *Int_TYPE* , *Real_TYPE*
VARIABLES *count*
INVARIANT $count \in INT$
INITIALISATION $count := 0$
OPERATIONS
$cc \longleftarrow$ **OpX** $\widehat{=}$
PCHOICE *1 // 2* **OF**
$count := count + 1 \parallel cc := count$
OR
$count := count - 1 \parallel cc := count$
END ;
OpY $\widehat{=}$ $count := 0$
END

Figure 4.3: The Counter Machine

We require that

$$\begin{aligned} e &\Rrightarrow [\mathit{init}]\, E \\ E &\Rrightarrow [\mathsf{OpX}]\, E \\ E &\Rrightarrow [\mathsf{OpY}]\, E, \end{aligned} \tag{4.6}$$

assures that

$$e \Rrightarrow [\mathsf{Init}; \mathsf{Op?}; \mathsf{Op?}; \ldots; \mathsf{Op?}]\, E \tag{4.7}$$

for any finite sequence $\mathsf{Op?}; \mathsf{Op?}; \ldots; \mathsf{Op?}$ of operations, no matter when or how they are chosen. (Recall that e is some initial expression, possibly depending on parameters and/or constants of the machine).

In contrast with the standard case, the "when or how" decision makes an important difference in the probabilistic world. In the following example, we show why it does.

Consider the *Counter* machine shown in Fig. 4.3. This machine *fails* to satisfy our probabilistic proof obligations (4.6), even though

$$0 \Rrightarrow [\mathit{init}; \mathsf{Op?}; \mathsf{Op?}; ...; \mathsf{Op?}]\, count \tag{4.8}$$

holds, for any finite sequence Op?; Op?; ...; Op? of operations *chosen in advance*. Here, the initial expression e is 0.

The machine fails to satisfy (4.6) because $count \Rrightarrow [\mathsf{OpY}]\, count$ cannot be proved. The reason that it *must* fail is that (4.8) is not true —for this machine— if the operations can be chosen on-the-fly. Consider for example the program fragment

$$\begin{array}{lll} Prog & \widehat{=} & \mathit{init}; \\ & & c \longleftarrow \mathsf{OpX}; \\ & & \textbf{IF } c = 1 \textbf{ THEN } \mathsf{OpY} \textbf{ ELSE } \mathsf{OpX} \textbf{ END} \,. \end{array} \tag{4.9}$$

The *IF*-statement represents a choice, on-the-fly, of whether to execute OpX or OpY as the second operation; and it is readily verified that the expected value of "count" after (4.9) is -0.5, which fails the instantiation (4.8) of the general proof obligation (4.6) for this machine. That is, we do *not* have $0 \Rrightarrow [Prog]\, count$.

Thus, the answer to the title of this section is

probabilistic invariants guarantee (4.7) provided (4.6) holds.

The constraint "no matter how the operations are chosen" in (4.7) seems to be too strong but it is absolutely necessary: the (usual) situation is that our machine must behave correctly no matter what the environment makes use of it. A system containing a fragment like (4.9) is a perfectly reasonable use of the Counter machine; any system that uses machine composition has access to all operations and the state of the component machines. The choice of operations is demonic and the above example shows that it is possible to choose operations "on the fly" in such a way as to have a significant effect on the expectation. However, such a choice is valid and the semantics cannot depend on uniform choice of operations; the semantics must require the expectation to increase monotonically regardless of of the chosen order of execution of operations, hence requirement (4.6).

4.2.5 A Probabilistic Invariant for the Library

In this section, we try to find the probabilistic invariant for our probabilistic library (Fig. 4.2 on page 80) by "informal" reasoning.

Practically, with the probabilistic library, we want to estimate the number of books lost, in particular, the upper bound of the number of books lost, since that will affect the cost of running the library. We believe that, informally, the expected

value of number of books actually lost is $pp \times loansEnded$, since the probability of losing a book when ending a loan is in fact pp. This informal reasoning leads to:

> the expected value of $pp \times loansEnded - booksLost$ is at least 0.

Thus we define $E \mathrel{\widehat{=}} pp \times loansEnded - booksLost$ to be the expected-value invariant of the probabilistic library machine. It can be seen that the initial value for E is 0 (established by the initialisation).

Recall from Sec. 4.2.4, which discussed the meaning of the probabilistic invariant, that if we take the expected value of E for many observations during the running of operations of the probabilistic library, then the average value will be at least 0. From this, we can conclude for our probabilistic library machine, the expected number of books lost (value of *booksLost*) is bounded above by $pp \times loansEnded$.

4.2.6 Proof Obligations

Recall from Sec. 2.3 that the proof obligations for a non-probabilistic (standard) machine are (where we ignore the obligations about the context of the machine):

N1: Under the context of the machine (information about parameters, sets and constants), the initialisation needs to establish the invariant:

$$[init]I \ .$$

N2: The operations need to maintain the invariant given that the precondition holds.

$$I \Rightarrow [Op]I \ .$$

For probabilistic machines, the same ideas will be applied, except that the invariant may now take *real* values instead of Boolean. In order to prove that the *real* invariant is bounded below, we have to prove the following:

P1: Under the context of the machine (information about parameters, sets and constants), the initialisation needs to establish the lower bound of the probabilistic invariant:

$$e \Rrightarrow [Init]E \ .$$

P2: Given that the precondition holds, the operations do not decrease the expected value of the probabilistic invariant, i.e. the expected value of the invariant after the operation is at least the actual value before the operation

$$E \Rrightarrow [Op]E \ .$$

We have to prove the above for each *real-valued* invariant. The standard invariants (Boolean-valued) can be treated the same as before (with probabilistic choice substitution being treated as demonic). Consequently, proof obligations for the *probabilistic* (expectation) and Boolean invariants may be generated, and proved, separately.

4.2.7 Proving the Obligations

Here, we only discuss the proof of maintenance of the probabilistic invariant:

$$E \quad \widehat{=} \quad pp \times loansEnded - booksLost \ . \tag{4.10}$$

The maintenance of the standard invariant is straightforward and similar to the standard library (with the probabilistic choice interpreted demonically).

In the example in Fig. 4.2 on page 80, consider the proof obligation for the initialisation (*P1*) [1]. We have to prove that

$$0 \Rrightarrow [init]E \ .$$

Consider the right-hand side of the inequality:

$$[init]E$$

$$\equiv \quad \left[\begin{array}{ll} booksInLibrary & := totalBooks \ || \\ loansStarted, & \\ loansEnded, & \\ booksLost & := 0,0,0 \end{array}\right] E \qquad \text{definition of } init$$

$$\equiv \qquad \text{definition of } E$$

$$\left[\begin{array}{ll} booksInLibrary & := totalBooks \ || \\ loansStarted, & \\ loansEnded, & \\ booksLost & := 0,0,0 \end{array}\right] \left(\begin{array}{c} pp \times loansEnded \\ -booksLost \end{array}\right)$$

$$\equiv \quad pp \times 0 - 0 \qquad \text{simple substitutions}$$

$$\equiv \quad 0 \qquad \text{arithmetic}$$

[1] All calculations use real numbers, but we will omit any type casting, e.g. $real(\cdot)$.

So we have shown that the initialisation establishes the initial lower bound for the probabilistic invariant E as required.

With the proof obligation *P2* for operation StartLoan, since the operation both increases $loansStarted$ and decreases $booksInLibrary$ deterministically, and since the real-valued invariant does not contain either variable, it can be easily proved that the operation maintains the invariant.

We have to do similar reasoning with the EndLoan operation, i.e. to prove that $E \Rrightarrow [EndLoan]E$ (proof obligation *P2*). Using $\overline{pp}$ for $1 - pp$, we calculate

$$[EndLoan]E$$

$\equiv$ definition of EndLoan

$$\left[\begin{array}{l} \left(\begin{array}{l} (booksLost := booksLost + 1 \\ {}_{pp}\oplus \\ booksInLibrary := booksInLibrary + 1) \end{array}\right) \\ || \\ loansEnded := loansEnded + 1 \end{array}\right] E$$

$\equiv$ parallel substitution with ${}_{pp}\oplus$, see below

$$\left[\begin{array}{l} \left(\begin{array}{l} (booksLost := booksLost + 1 \\ || \\ loansEnded := loansEnded + 1) \end{array}\right) \\ {}_{pp}\oplus \\ \left(\begin{array}{l} (booksInLibrary := booksInLibrary + 1 \\ || \\ loansEnded := loansEnded + 1) \end{array}\right) \end{array}\right] \left(\begin{array}{c} pp \times loansEnded \\ -booksLost \end{array}\right)$$

$\equiv$ probabilistic choice substitution ${}_{pp}\oplus$

$$pp \times \left[\begin{array}{l} booksLost := booksLost + 1 \\ || \\ loansEnded := loansEnded + 1 \end{array}\right] \left(\begin{array}{c} pp \times loansEnded \\ -booksLost \end{array}\right)$$

$$+$$

$$\overline{pp} \times \left[\begin{array}{l} booksInLibrary := booksInLibrary + 1 \\ || \\ loansEnded := loansEnded + 1 \end{array}\right] \left(\begin{array}{c} pp \times loansEnded \\ -booksLost \end{array}\right)$$

$\equiv$ parallel substitution and simple substitution

$$\begin{array}{ll} & pp \times (pp \times (loansEnded + 1) - (booksLost + 1)) \\ + & \overline{pp} \times (pp \times (loansEnded + 1) - booksLost) \end{array}$$

$\equiv$ $pp \times loansEnded - booksLost$ arithmetic

$\equiv$ E

For the deferred judgement, we use the distribution property of parallel substitution with respect to probabilistic choice substitution, i.e. that if we have U is a deterministic substitution then

$$(S \ {}_p\!\oplus\ T) \,||\, U \quad \equiv \quad (S||U) \ {}_p\!\oplus\ (T||U)\,.$$

So we have shown that $E \Rrightarrow [EndLoan]E$. (In fact, the expectation is unchanged, i.e. $E \equiv [EndLoan]E$ since there is no demonic non-determinism.)

In this example, we have specified a library system that includes the chance of books being lost. From the probabilistic invariant, provided that we know pp, we can estimate the cost of maintaining the library (relating to the number of books lost). Furthermore, we have discussed how we can reason about the specification and how to write it in *pB* (or rather *pAMN* for that matter).

4.2.8 What the Invariant Means

With the two calculations in the previous section, we have established the mathematical validity of the (probabilistic) invariant E for the machine in Fig. 4.2 on page 80, in the sense that the proof obligations are satisfied. How do we interpret this validity?

Recall Sec. 4.2.4: it means that over a large number of tests of the machine, carried out by an adversary who can choose to resolve demonic choice within operations any way that he wishes (although there are none in our example), and who can choose to invoke operations in any order (i.e. to resolve demonic choice "between" operations), we will observe that the average value of E is at least the stated value.

For the case of the machine in Fig. 4.2 on page 80, we conclude therefore that the expected value of $pp \times loansEnded - booksLost$ is at least 0; no matter what the adversary does. We wrote the invariant this way around (e.g. instead of

$totalCost \longleftarrow$ **StockTake** $\widehat{=}$

BEGIN

$totalCost := cost \times booksLost \parallel$

$booksInLibrary := booksInLibrary + booksLost \parallel$

$loansStarted := loansStarted - loansEnded \parallel$

$loansEnded := 0 \parallel$

$booksLost := 0$

END

Figure 4.4: StockTake operation

its negation) so that we could give an expected upper bound for $booksLost$—it is $pp \times loansEnded$ always.

In general, we might wish to establish several such average-case inequalities. For each one we would formulate a suitable probabilistic invariant and lower bound; and each would generate its own proof obligations (*P1*), (*P2*) as above.

4.3 Pitfalls: Mixing Demonic and Probabilistic Choice

With the validity of a probabilistic invariant, we assure that there is a lower bound for its expected (average) value over many runs of machine-operation invocations. In this section, we give a clearer picture of how strong this requirement is, by using an example in which an adversarial user of the system has complete freedom in choosing (on-the-fly) which operations to execute. The mathematical reasoning about maintaining the probabilistic invariant guides us in the design of machines that are well-behaved even against such adversaries.

4.3.1 StockTake Breaks the Probabilistic Invariant

For a real library, imagine that it needs to do a stocktake annually, i.e. update the number of books lost, and reset the information of the library. At the same time, the library wants to estimate the running cost for that particular year. For simplicity, we assume that the cost for replacing a book in the library is a constant, $cost$. The operation StockTake is defined in Fig. 4.4.

The StockTake operation is similar to the initialisation, but with an extra output to represent the cost for replacing the books lost. One can easily prove that the operation maintains the standard invariant. The surprise comes when trying to prove the obligation for maintaining the probabilistic invariant by this operation. We have to prove that $E \Rrightarrow [StockTake]E$. Consider the right-hand side of that inequality (only considering the effects on variables $loansEnded$ and $booksLost$, which are related to the value of E):

$$
\begin{array}{lll}
 & [StockTake]E & \\
\equiv & [loansEnded, booksLost := 0, 0]E & \text{definition of StockTake} \\
\equiv & & \text{definition of } E \\
 & [loansEnded, booksLost := 0, 0](pp \times loansEnded - booksLost) & \\
\equiv & pp \times 0 - 0 & \text{simple substitutions} \\
\equiv & 0\,. & \text{arithmetic}
\end{array}
$$

So in order to show the (probabilistic) invariant does not decrease we must prove that

$$pp \times loansEnded - booksLost \quad \Rrightarrow \quad 0\,, \tag{4.11}$$

which we cannot prove in this context. The question here is what did we do wrong in the above operation.

4.3.2 Surprising Interaction of Demonic and Probabilistic Choice

To understand the failure to maintain the probabilistic invariant we will discuss a number of aspects.

Initialisation is not forever The reason the StockTake operation is added to the library firstly is to have more details for the library. It is not the intention for this operation to cause any problems with the probability. It turns out however that our hopes for the operation StockTake is too optimistic. As indicated earlier, the operation is very similar to the machine initialisation. Starting from the intuition in standard *B*, the initialisation can be invoked many times whenever we want, and clearly the standard invariant will be maintained (and this fact can be easily proved). However, as stated in Sec. 4.2.4, the meaning of maintaining probabilistic invariant is that its expected value does not decrease. If we view the standard

B invariant in this particular way, it is a Boolean expression that is must evaluate to true (1) after the initialisation. Moreover, this also represents a monotonic increase over its value before initialisation, which was either false (0) or true (1). Of course, during the execution, if we re-run the initialisation as a normal operation, it starts from an expected value of true (1) and so still guarantees not to decrease the expected value.

For real-value expectations, the expected values of the probabilistic invariant can have values other than just the two discrete values 0 or 1. This results in a strong obligation to maintain the expected value, and as a consequence, some notions taken from the simpler Boolean context will not be valid.

According to our interpretation, the initialisation of a *pB* machine establishes the probabilistic invariant on the assumption of a lower bound of the expectation; e.g. 0 for the probabilistic library. For any run, the sequence of operations monotonically increases the expected value of the probabilistic invariant; hence it is presumptuous to expect that the initialisation would maintain the expected value of such invariant if run again at an arbitrary time. Thus, if an operation duplicates the initialisation of a machine, there are no guarantees for this operation to maintain the probabilistic invariant.

The effect of demonic non-determinism It is worth reminding ourselves that there are two types of demonic non-determinism in *B*. The first form of demonic non-determinism is explicit within operations in terms of non-deterministic choice substitution ([]). The second form is implicit in terms of the choice of invoking operations for a machine. As a consequence, the machines must be designed to guarantee that any undesirable operation sequences do not lead to a violation of critical properties in machine behaviour. This is also the precise reason why we use invariance for both non-probabilistic and probabilistic machines: to establish that such critical properties are maintained regardless of the choice of operation sequence.

In our example, the probabilistic library machine is designed to establish an upper bound of $pp \times loansEnded$ for $booksLost$. Prior to the addition of the StockTake operation, this was being controlled solely by the probabilistic choice within the EndLoan operation (the StartLoan operation does not affect the value of the probabilistic invariant). By introducing the StockTake operation, there is

an opportunity for demonic non-determinism (between operations) to subvert that expectation. Consider the following scenario where a malevolent library administrator wishes to show that the library loan system is "broken" by trying to show that the actual rate of book loss is higher than the desired value pp. He or she can choose a policy of running StockTake only when the value of $booksLost$ is large relative to $pp \times loansEnded$, a situation that is guaranteed to with probability one to occur at some points: if the library manager is "encouraged" to look only at those points then he will indeed believe that the system is "broken".

Practically, during the operation of the probabilistic library, there can be times when loss rate will be higher than expected. The above scenario exploits the fact that operations can be chosen demonically to run only at those times. In order to avoid this, we might suggest that, when testing a system, a machine-tester, in selecting what operation to run next, should not be able to look at the current state of the machine. That this is reasonable can be seen by testing a coin: it would be wrong to flip the coin until heads shows, and then say "look, it always gives heads".

The demonic choice taken by the tester as to which operation to run next is called *oblivious* if he is not allowed to see the machine's state. Obviously, this takes away from the tester some of the power compared to when he is allowed to look at the machine state (in this case, we say that it is *omniscient*). Given that the tester has more power to control the execution of the machine, omniscient testing is clearly more severe than oblivious testing. Hence our proof obligations, which are sufficient for correctness under omniscient testing, are stronger than the other alternative obligations we might consider to guarantee the correctness under oblivious testing. In fact, if we formulate the proof obligations for oblivious testing, operation StockTake would probably be admitted (since a malevolent library administrator will not know when the actual $booksLost$ is high relative to $pp \times loansEnded$). [2]

Finally, when it comes to standard machines, omniscient and oblivious testing makes no difference. If a standard machine is guaranteed to pass oblivious testing, then it is also guaranteed to pass omniscient testing. Only probabilistic [3] machines make the distinction between omniscient and oblivious testing.

[2] The formulation of oblivious testing turns out to be surprisingly complex. The notion of what "can be seen" is hard to formalise, and we believe that it would limit the way machine can be composed (to be consistent with "oblivious"). We regard this as a fruitful line of further research.

[3] Adding angelic choice also reveals the distinction: any two of probabilistic/angelic/demonic are sufficient.

4.3.3 Capturing Long-term Behaviour

The failure when proving the obligations for the StockTake operation is in fact (mathematically) suggesting what we should do to fix it. We introduce a new variable called fix in order to be able to prove the obligation. Routine calculation tells us that we must change the machine as follows: initially, fix is given the value 0; fix is unchanged in StartLoan and EndLoan operations; and in the StockTake operation, we modify fix to maintain the information about the number of $booksLost$ related to $pp \times loansEnded$, which is crucial for the expectation:

$$fix := pp \times loansEnded - booksLost + fix \ . \tag{4.12}$$

We also need to change our probabilistic invariant accordingly:

$$E' \ \widehat{=} \ pp \times loansEnded - booksLost + fix \ . \tag{4.13}$$

The lower bound for our new expectation is 0 also, which is established trivially by the initialisation. For operations StartLoan and EndLoan, the value of fix does not change so the probabilistic invariant E' is maintained (similar to those original cases, but with fix acting as a constant). With the StockTake operation, we can prove the maintenance of the probabilistic invariant E' as follows (we only consider the changes for the set of variables relating to the value of E', i.e. $loansEnded$, $booksLost$ and fix):

$$
\begin{array}{ll}
 & [StockTake]E' \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of StockTake} \\
 & \left[\begin{array}{lll} loansEnded & := 0 & || \\ booksLost & := 0 & || \\ fix & := pp \times loansEnded - & \\ & \quad booksLost + fix & \end{array}\right] E' \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{definition of } E' \\
 & \left[\begin{array}{lll} loansEnded & := 0 & || \\ booksLost & := 0 & || \\ fix & := pp \times loansEnded - & \\ & \quad booksLost + fix & \end{array}\right] \left(\begin{array}{c} pp \times loansEnded - \\ booksLost + fix \end{array}\right) \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{simple substitutions} \\
 & pp \times 0 - 0 + (pp \times loansEnded - booksLost + fix) \\
\equiv & \quad pp \times loansEnded - booksLost + fix \qquad\qquad\qquad\qquad \text{arithmetic}
\end{array}
$$

So we prove that $E' \Rrightarrow [StockTake]E'$, i.e. the StockTake operation does not decrease the expected value of E'.

What is the meaning of the variable fix that we have introduced? In fact, it records the long-term surplus/deficit indicator of the books lost in the library compared with what we expected to lose.

Moreover, when we re-initialise variables of the library as in the original version of StockTake, we lost some information about the number of books actually lost. The new probabilistic invariant repairs that by forcing us not to "lose information" that way. The reason for this is the long-term behaviour can only be captured if there are variables which record this information; any operation that somehow "deletes" the long-term behaviour will violate the proof obligations. This is in fact what probability is about.

In our probabilistic library, we can have a case in which —without fix— a hostile library administrator could decide (the demonic choice between operations) to run StockTake only when the rate of books lost was "running high", and by doing so, give a false picture of the "current snapshot" of the long-term behaviour of the library. Having fix makes sure that any snapshot includes *every* behaviour up to that point.

4.4 Actual Changes to the *B-Toolkit*

The *B-Toolkit* needs to be changed in order to support the new construct and syntax of probabilistic machines. Moreover, the new toolkit should be capable of generating proof obligation for the maintenance of probabilistic invariants. The extension from *GSL* to *pGSL* (hence the necessary extension from *AMN* to *pAMN*) is by introducing the probabilistic-choice substitution. This introduction and the new probabilistic invariant affect the processing of probabilistic machines (such as analysing, type checking, proof-obligation generating and proving).

Firstly, since we use real numbers for our probabilistic choice and probabilistic invariant, we introduce real numbers by using the Real_TYPE context machine, as in Sec. 3.4. Recall from the Real_TYPE machine that currently it provides the set of non-negative rational numbers, where numbers are denoted by constructors $frac(m, n)$.

In this section, we introduce two new *pAMN* constructs: the EXPECTATIONS clause to declare probabilistic invariants and the corresponding lower-bounds; and the *PCHOICE-OF-OR* clause for probabilistic choice substitution.

With the introduction of new *pAMN* clauses, the analyser of the *B-Toolkit* needs to be modified to accept the new constructs (including parsing, type checking) and translate the *pAMN* into a canonical, syntactically parsed form. The type information is also stored in canonical form in order to be used in later phases.

The most important change to the *B-Toolkit* is with the proof obligation generator. The new toolkit needs to generate obligations for the new *PCHOICE* clause and for the maintenance of probabilistic invariants. For probabilistic invariants, the proof obligations must follow the semantics of *pGSL*, as stated in figure Fig. 2.10 on page 36. Assume that we have a probabilistic machine as follows:

MACHINE MachineName(x)
CONSTRAINTS P
CONSTANTS c
PROPERTIES Q
VARIABLES v
INVARIANT R
EXPECTATIONS $e_1 \Rrightarrow E_1; \ldots; e_n \Rrightarrow E_n$
INITIALISATION T
OPERATIONS
  z $\longleftarrow$ OpName $\widehat{=}$ **PRE** L **THEN** S **END**;
  ...
END .

Beside the normal proof obligations generated for maintaining the standard invariant (as described in Sec. 2.4 [4]), each pair of expectation and its lower-bound $e_i \Rrightarrow E_i$ yields the following proof obligations:

1. $P \wedge Q \Rightarrow e_i \Rrightarrow [T]\, E_i$
2. $P \wedge Q \wedge R \wedge L \Rightarrow E_i \Rrightarrow [S]\, E_i$

The first condition states that the lower-bound for each expectation is established by the initialisation and the second condition states that the expectation is maintained (non-decreased) by each operation given the context information, the invariant and the precondition of the operation.

For practical reasons, we leave Boolean expressions unchanged, even though they could be converted to numeric expressions (using embedded predicate $\langle\cdot\rangle$).

[4] When proving obligations related to the standard invariant, probabilistic choice is treated demonically.

This has the effect of ensuring that the proof of all Boolean goals or sub-goals will proceed using the standard proof rules.

It turns out that no changes were required for the provers, but some new proof rules were needed to support real-number evaluations. This is a consequence of having probabilistic invariants (which are real numbers).

Finally, some small changes ware required to "mark-up" the EXPECTATIONS, *PCHOICE* clauses and the $\Rrightarrow$ expectation order.

An advantage of separating the canonical form and type information after the analysing phase of the Toolkit is that, afterwards, all other phases can be based purely on syntax. This, somewhat surprisingly, simplified the modification of the *B-Toolkit* to be able to process numeric, rather than Boolean logic since, after analysing, proof obligations and proof rules are typeless. In the new context including real numbers, some proof rules became invalid and had to be modified. Moreover, some new rules had to be added to support the reasoning about real numbers. Currently, the probabilistic information (of expectations) of a probabilistic machine is stored separately from the unaltered standard (non-probabilistic) information, but in general, they could be merged.

In our example of the probabilistic library, the ProbabilisticLibrary machine has been analysed, proof obligations have been generated and discharged; and the machine has been marked up using the modified *B-Toolkit*.

4.5 Conclusions and Future Work

We have presented a practical approach to extending *B* to include probability. We have extended *B* machine to include probabilistic choice constructs and probabilistic invariants. New proof obligations for the establishment and maintenance of probabilistic invariants have been described. We have applied our theoretical work to a simple case study of a library, in which books may be (occasionally) lost. We have used this example to illustrate how to write programs in *pB* (using *pAMN* notation) and how to reason formally about their correctness. Lastly, we have shown that there are significant differences between standard *B* constructs and *pB* constructs, differences that need to be carefully understood.

The *B-Toolkit* has been modified to incorporate the new *pAMN* constructs and also to provide the generation and discharging of proof obligations. In some cases,

we have carefully designed our constructs to accommodate existing capabilities of the *B-Toolkit*.

This chapter discusses the first step of reasoning about "absolute" correctness for probabilistic system. This is in contrast with the "almost-certain" correctness that we have introduced in the previous chapter, where probability is deliberately introduced into the system to guarantee termination, even when the standard method fails.

With the introduction of probabilistic machines, we can model many performance properties of systems including "the expected time to achieve a stated goal" [39] or the "probability that a goal will be achieved within a specific time". Expected time to achieve "stability" for instance is of particular importance when systems use probability as an "in-built" facility.

There are some other tools, such as model checker PRISM [55], which can also deal with these problems. However the model-checking approach contrasts fundamentally with *B* in that model-checkers are analysis, rather than design tools.

The introduction of *Event-B* [3, 6] gives another possible approach. In *Event-B*, each operation (or now called an "event") has a naked guarded command with a simple body. If we want to introduce probability into *Event-B*, then naturally the probability should be somehow integrated into the naked guard. With this approach, the body of operations can be kept as simple as possible (without probability), but the reasoning about the correctness of probabilistic guard requires other considerations that will be not discussed in this dissertation. More information about this approach can be found in [46].

Chapter 5

Probabilistic Specification Substitutions

5.1 Refinement of Probabilistic Systems

In earlier chapters, we introduced a number of extensions to *B* in order to incorporate what could be called "low-level" features. The results of that work are a method (and a corresponding toolkit) for probability-one termination, called *qB*, and a simple method for specifying probabilistic program with full probabilistic reasoning, called *pB*. In the former method, *qB* would be applied to the final stages of an algorithm like the IEEE 1394 (FireWire) protocol [5, 17] or Rabin's Choice-Coordination [57] where in each case a potential livelock is resolved with probability one. In the latter method, *pB* has the full capability to reason about probabilistic systems (not just probability-one termination) via the introduction of probabilistic choice substitutions.

The probabilistic-choice substitution can be considered as "code" in the sense that it is directly translatable into executable code. It will usually be used in the later stage of software development (even though we have introduced and used it in specifications from the outset).

In this chapter, we change our focus and concentrate on the "high-level" issues of the probabilistic-system development. We want to start from abstract specifications (usually containing some form of non-determinism), and then move through the concept of "refinement" to reach the implementations of such specifications.

Because of its abstraction, the specification needs to be refined (possibly through many steps) into a more concrete implementation before being translated into code. This is the backbone of software development using *B*.

We propose a new construct called "probabilistic specification substitution" which is the *B* version of the *probabilistic specification substitution* for *Z* [67]. We extend the "fundamental theorem" for the traditional specification statement [42] to the probabilistic context, which allows us to show that the new substitution is "valid" within the refinement context.

Furthermore, in order to be able to implement such specifications, we extend the methods for reasoning about correctness of standard loops to probabilistic loops, using *probabilistic wp-logic* [49]. We focus on the practical side of the problem, in which we can separate standard (Boolean) and probabilistic (numerical) reasoning. This separation allows us to extend the *B-Toolkit* without much change to the original Boolean logic underneath. The new (complex) obligations for probabilistic properties can be generated and proved (separately) with some extra (simple) modifications to the prover, while standard reasoning can use those facilities available within the original toolkit.

We use the well known example of the "Min-Cut" algorithm to show how the new construct can be used. We start by specifying the algorithm using probabilistic specification substitution, then continue by implementing the algorithm using probabilistic loops, and we give the reasoning for the correctness of the development. We also give some explanation about an unexpected issue that arises during the development of the Min-Cut algorithm.

The chapter is structured as follows: in Sec. 5.2 we first review the traditional specification substitution for standard systems, and we also introduce the probabilistic specification substitution to describe probabilistic systems; in Sec. 5.3 we appeal to the expectation semantics from *pGSL* and obtain the probabilistic fundamental theorem which is a generalised version of the corresponding standard theorem. For practical reasons, in the chapter, we consider the set of expectations which are within the closed interval $[0..1]$.

In Sec. 5.4 we discuss proof obligations for probabilistic loops, which are the generalisation of the variant and invariant technique for standard loops.

In Sec. 5.6 we first apply the fundamental theorem to the Min-Cut algorithm,

and we also set out the proof obligations for verifying the refinement relationship, which will help us to find an unexpected issue related to our probabilistic specification substitution.

In Sec. 5.7, we introduce the *terminating* probabilistic specification substitution and the corresponding fundamental theorem, This is to take advantage of the fact that obligations about termination are always checked when developing systems using the *B-Toolkit*. Finally, we summarise the development, draw our conclusions and outline possible future work.

The example of the Min-Cut algorithm, which is used in the chapter, has been developed and completely proved using the modified *B-Toolkit*.

5.2 Probabilistic Specification Substitutions

Specification is the starting point for the refinement steps that lead to executable code. This is the main framework for *B*, whether standard or probabilistic. Here, we turn to probabilistic specification substitutions, which will be used as specifications for probabilistic developments. We begin by reviewing the standard specification substitution that *B* already contains.

5.2.1 Standard Specification Substitutions

In this section, we briefly review the interaction of specifications and so-called "specification substitutions". During the specification stage, a standard (i.e. non-probabilistic) specification S is traditionally specified by giving a (not necessarily weakest) precondition P and a post-condition Q, where both P and Q are predicates over the state space:

$$P \quad \Rightarrow \quad [S]\, Q\,. \tag{5.1}$$

Informally, this means that, assuming an initial state satisfying P, the execution of S must establish a final state satisfying Q, which directs an implementer to develop a program with the required property. For the purpose of abstraction, it is common to use pre- and post-conditions to describe the possible behaviour of the system to be built in specification stage of a development. There are many forms of this, which are summarised below:

- Hoare triples [26]:

$$\{P\} \quad S \quad \{Q\} \tag{5.2}$$

where P and Q are predicates over the program state, and S is program statement. If P holds in the initial state and S is executed and terminates, then Q is guaranteed to hold in the final state.

- Dijkstra's weakest precondition [16]:

$$P \quad \Rightarrow \quad wp(S, Q) \tag{5.3}$$

where P and Q are predicates over the program state, and S is program statement. The notation $wp(S, Q)$ denotes the weakest precondition of S with respect to Q, i.e. the weakest precondition, under which Q is guaranteed to be established after the execution of S.

- Morgan's specification statements [43]:

$$v : [P, Q] \tag{5.4}$$

where v is the *frame*, a sub-vector of the program variables whose values may change. P and Q are predicates describing the initial state and the final state, respectively.

Other notations also include Morris's prescription [52] and Back's pre/post-condition specifications [8].

In *B* (or more precisely, *GSL*) [4], the same idea is presented with a different syntax:

$$P \mid v : Q \,, \tag{5.5}$$

with the meaning that the substitution will establish Q under the precondition P, and change only the variables in v. In this form, we will always assume that P and Q are predicates over x and over x_0, v, respectively. The variables v (sub-set of the variables x) are those that can be possibly changed by the substitution. The variables x_0 are distinct from x and represent their original values. Intuitively, the meaning of (5.5) can be derived from the semantics presented Fig. 2.2 on page 12 by decomposing it into an unbounded choice substitution and a precondition substitution as follows:

$$P \mid @\, v' \cdot \big(\big[x_0, v := x, v'\big]\, Q \implies v := v'\big) \,.$$

In fact, this is the form, into which the *B-Toolkit* will de-sugar the substitution. It is common in *B* to get the meaning of a substitution by decomposing it in this way. When this is done, it can be easily seen that specification (5.5) is just another presentation of those illustrated in (5.2), (5.3) or (5.4).

5.2.2 Probabilistic Specification Substitutions

The ideas in Sec. 5.2.1 can be generalised to the probabilistic context. In this section, we propose a probabilistic version of (5.5) which has the same role in the probabilistic world as the original standard specification substitution does in the standard world.

Recall that in the expectation logic, we write

$$A \quad \Rrightarrow \quad [S]B \ , \tag{5.6}$$

to mean that the execution of S guarantees to establish that the expected value of B over the final state distributions is bounded below by the value of A in the initial state. By analogy with the connection between Dijkstra-style wp-specification and Morgan's specification statement, we propose a probabilistic specification substitution written as in the standard case, that is

$$A \ \mid \ v : \ B \ , \tag{5.7}$$

except that now A is an expectation defined over the program variables, B is an expectation that may additionally refer to x_0 and v. The variables v (which we also call the *frame* of the substitution) are a sub-vector of program variables x, which the substitution "constrains" to change only those variables (we will address what we mean by *constrains* later).

As the first example, if we want to specify a coin that comes up heads with probability at least one-half, then in the style of (5.6) we would write

$$\frac{1}{2} \quad \Rrightarrow \quad [\mathit{Flip}]\langle \mathit{coin} = \mathit{Head} \rangle \ ,$$

where coin is the state variable with possible values $\{\mathit{Head}, \mathit{Tail}\}$. In the style of (5.7), we would instead specify the substitution Flip as

$$\frac{1}{2} \ \mid \ \mathit{coin} : \ \langle \mathit{coin} = \mathit{Head} \rangle \ , \tag{5.8}$$

for the following reason: it achieves $coin = Head$ (post-expectation $\langle coin = Head \rangle$) with probability at least $\frac{1}{2}$ (pre-expectation).

Thus the *probabilistic specification substitution* generalises the traditional standard specification substitution into the probabilistic program domain.

We now give the definition for (5.7) so that we can explain why specifications like *Flip* have the meaning we claim for them. The precise semantics of our new construct is as follows.

Definition 5 *The semantics of the specification substitution "$A \mid v : B$", with respect to arbitrary post-expectation E (containing no x_0) is given by*

$$[A \mid v : B]E \quad \widehat{=} \quad A \times [x_0 := x] \sqcap x \cdot (E \div B^w) \ , \tag{5.9}$$

where x is the vector of all variables appearing in A, B or E, w is the vector of variables in x but not in v, and

$$B^w \quad \widehat{=} \quad B \times \langle w = w_0 \rangle \ .$$

The symbol $\div$ denotes division of real numbers [1].

In general, $\sqcap x \cdot (E)$ means the greatest lower bound of the expression (expectation) E over the possible values of x. The scope of the minimum is indicated by the brackets that follow the "$\cdot$", and hence the expression $\sqcap x \cdot (E \div B^w)$ means the minimum of "$E \div B^w$" over all x.

Intuitively, Def. 5 says that the specification takes an initial state to any one of a number of final state distributions, all of which satisfy the requirement that the expected value of B evaluated over that final distribution is bounded below by A evaluated on the initial distribution. Given this and applying the scaling property of a probabilistic program, the definition calculates the expected value of an arbitrary expectation E (instead of B). The scaling property in [49] states the following: multiplication by a non-negative constant distributes through substitution, i.e.

$$[S]\,(c \times E) \quad \equiv \quad c \times [S]\,E \ , \tag{5.10}$$

for any substitution S, any post-expectation E, and any constant c. We use B^w in the division (instead of B) since the variables in w are "constrained" not to change.

[1] We assume that $n \div 0 = \infty$ for any number n.

We will address the issue of variables that are allowed to change in the next chapter. In our definition, $\sqcap x \cdot (E \div B^w)$ is the biggest constant factor between E and B^w. The substitution $x_0 := x$ just ensures that the variables in the original state (zero-subscripted) are converted to the state variables.

Taking the example of Prog. (5.8), we can calculate the probability that the outcome is heads, i.e. with respect to the post-expectation $\langle coin = Head \rangle$. From Def. 5 it is given as

$$
\begin{array}{lll}
 & \left[\frac{1}{2} \mid coin : \langle coin = Head \rangle\right] \langle coin = Head \rangle & \\
\equiv & & \text{Def. 5} \\
 & \frac{1}{2} \times [coin_0 := coin] \sqcap coin \cdot (\langle coin = Head \rangle \div \langle coin = Head \rangle) & \\
\equiv & \frac{1}{2} \times [coin_0 := coin]\, 1 & \text{arithmetic}\,[2] \\
\equiv & \frac{1}{2}\,. & \text{arithmetic and simple substitution}
\end{array}
$$

So indeed the probability that Prog. (5.8) establishes $coin$ is $Head$ is at least $\frac{1}{2}$.

If we calculate the probability that the outcome of the same program is tails, however, we have

$$
\begin{array}{lll}
 & \left[\frac{1}{2} \mid coin : \langle coin = Head \rangle\right] \langle coin = Tail \rangle & \\
\equiv & & \text{Def. 5} \\
 & \frac{1}{2} \times [coin_0 := coin] \sqcap coin \cdot (\langle coin = Tail \rangle \div \langle coin = Head \rangle) & \\
\equiv & \frac{1}{2} \times [coin_0 := coin]\, 0 & \text{minimum } 0 \div 1 \text{ occurs at } coin = Head \\
\equiv & 0\,. & \text{arithmetic and simple substitution}
\end{array}
$$

The conclusion is that: Prog. (5.8) does not give any guarantee at all that the outcome is tails. We will address this point later, in Sec. 5.7.

5.3 The Fundamental Theorem

The substitution that we introduced in the last section can be used to specify a probabilistic system, but it is not automatically translatable into code. In other words, we have to refine such substitutions, ourselves, into more concrete constructs and —while doing so— prove the refinement relationship between them. In this section, we give justification to the semantics of the new substitution by looking at

[2] We assume that $x \div 0$ is ∞ for any x so that the $\sqcap$ ignores it.

the fundamental theorem that such semantics should obey. There is also a standard fundamental theorem, and we will propose the corresponding probabilistic version of it.

5.3.1 The Standard Fundamental Theorem

The standard fundamental theorem comes from the refinement calculus [43, 42]; here we explain it in terms of *B*-style notation.

Theorem 4 *Let "$P \mid v : Q$" be defined as in (5.5) and T be any program written in* GSL *with state variables x. Then*

$$(P \mid v : Q) \quad \sqsubseteq \quad T$$

if and only if

$$P \quad \Rightarrow \quad [x_0 := x][T]Q^w \ ,$$

where

$$Q^w \quad \widehat{=} \quad Q \ \wedge \ w = w_0 \ .$$

Similar theorems and their proofs can be found in [50, 7]. The theorem states that T guarantees to establish the post-condition Q in the after state if the initial state satisfies the precondition P and, in doing so, T changes only variables in v (and leaves variables in w unchanged). Hence, obviously, T satisfies the specification "$P \mid v : Q$".

5.3.2 The Probabilistic Fundamental Theorem

In this section, we give the probabilistic generalised version of the fundamental theorem, with respect to the expectation semantics from *pGSL*. We will apply the theorem to some simple examples to show how it can be used.

The probabilistic fundamental theorem can be stated as follows:

Theorem 5 *Let "$A \mid v : B$" be defined as in Def. 5 and T be any substitution which is free from variables x_0. Assume B satisfies the assumption:*

$$\forall\, x_0 \cdot (\exists\, v \cdot (B \neq 0)) \ . \tag{5.11}$$

Then

$$(A \mid v : B) \sqsubseteq T \quad \textit{iff} \quad A \ \Rrightarrow \ [x_0 := x][T]B^w \ .$$

Proof. We now prove the theorem in each direction separately using Lem. 6 and Lem. 7 below.

Lemma 6 *Let "$A \mid v : B$" and T be the same as in Thm. 5. If*

$$(A \mid v : B) \quad \sqsubseteq \quad T$$

then we have

$$A \quad \Rrightarrow \quad [x_0 := x][T]B^w \ ,$$

where, as before, x is "all variables", i.e. those occurring in A, B or T.

Proof. We begin the proof from the right-hand side. The first few calculations in the proof are because the post-expectation in Def. 5 (E there but B^w here) is not supposed to contain any occurrences of x_0.

$$[x_0 := x][T]B^w$$

$$\equiv \quad [x_0 := x]([T]B^w) \qquad \text{sequential substitution}$$

$$\equiv \quad [x_0 := x]([x' := x_0][T][x_0 := x']B^w) \qquad \begin{array}{r} x' \text{ are fresh variables} \\ T \text{ contains no } x_0 \end{array}$$

$$\equiv \quad [x_0 := x][x' := x_0]([T][x_0 := x']B^w) \qquad \text{sequential substitution}$$

$$\equiv \quad [x' := x]([T][x_0 := x']B^w) \qquad \text{no } x_0 \text{ in } [T][x_0 := x']B^w$$

$$\equiv \quad [x' := x]([T]([x_0 := x']B^w)) \qquad \text{sequential substitution}$$

$$\Leftarrow \qquad \text{monotonicity and refinement assumption}$$

$$[x' := x]([A \mid v : B]([x_0 := x']B^w))$$

$$\equiv \qquad \text{from Def. 5}$$

$$[x' := x]\,(A \times [x_0 := x] \sqcap x \cdot ([x_0 := x']B^w \div B^w))$$

$$\equiv \quad A \times [x' := x][x_0 := x] \sqcap x \cdot ([x_0 := x']B^w \div B^w) \qquad \text{no } x' \text{ in } A$$

$\equiv$ $[x_0 :=\ x]$ is free of x'

$A\ \times\ [x_0 :=\ x][x' :=\ x_0]\ \sqcap x \cdot ([x_0 :=\ x']B^w\ \div\ B^w)$

$\equiv$ $A\ \times\ [x_0 :=\ x]\ \sqcap x \cdot ([x' :=\ x_0]([x_0 :=\ x']B^w\ \div\ B^w))$ properties of $\sqcap$

$\equiv$ simple substitution and B^w is free of x'

$A\ \times\ [x_0 :=\ x]\ \sqcap x \cdot ([x' :=\ x_0][x_0 :=\ x']B^w\ \div\ B^w)$

$\equiv$ $A\ \times\ [x_0 :=\ x]\ \sqcap x \cdot (B^w\ \div\ B^w)$ sequential substitution
no x' in B^w

$\equiv$ $A\ \times\ [x_0 :=\ x]\ 1$ non-zero assumption (5.11) on B and $B^w \mathrel{\widehat{=}} B \times \langle w = w_0 \rangle$

$\equiv$ $A\ ,$ arithmetic

which completes the proof.

Lemma 7 *Let "$A\ \mid\ v :\ B$" and T be the same as in Thm. 5. If*

$$A \quad \Rrightarrow \quad [x_0 :=\ x][T]B^w \tag{5.12}$$

then we have

$$(A\ \mid\ v :\ B) \quad \sqsubseteq \quad T\ .$$

Proof. We begin by calculating the application of substitution "$A\ \mid\ v :\ B$" to any expectation E that is free from x_0:

$[A\ \mid\ v :\ B]E$

$\equiv$ $A\ \times\ [x_0 :=\ x]\ \sqcap x \cdot (E\ \div\ B^w)$ Def. 5

$\Rrightarrow$ $[x_0 :=\ x][T]B^w\ \times\ [x_0 :=\ x]\ \sqcap x \cdot (E\ \div\ B^w)$ Assumption (5.12)

$\equiv$ $[x_0 :=\ x]\,([T]B^w\ \times\ \sqcap x \cdot (E\ \div\ B^w))$ simple substitution $[x_0 :=\ x]$

$\equiv$ T free from x_0 and scaling $[T]$; see below

$[x_0 :=\ x]([T]\,(\sqcap x \cdot (E\ \div\ B^w) \times B^w))$

$\Rrightarrow$ monotonicity
$\sqcap x \cdot (E \div B^w) \Rrightarrow E \div B^w$ as E free from x_0

$$[x_0 := x]([T]((E \div B^w) \times B^w))$$

$\equiv$ $[x_0 := x]([T]E)$ non-zero assumption on B

$\equiv$ $[T]E$. both T and E free from x_0

Since E was arbitrary, we have that

$$(A \mid v : B) \sqsubseteq T ,$$

which completes the proof.

For the deferred judgement, we use the scaling property (5.10) of substitutions which states that multiplication by a non-negative constant distributes through substitutions [49].

As an example for the fundamental theorem, we return to the specification of a coin in Prog. (5.8). We look at some programs that refine this specification.

- $T_1 \;\widehat{=}\; coin := Head \;{}_{\frac{1}{2}}\!\oplus\, \mathsf{abort}$.

  This program returns a $Head$ with probability one-half, otherwise, it does not even guarantee to terminate. We prove the refinement by considering:

  $$[coin_0 := coin][T_1]\langle coin = Head\rangle$$

  $\equiv$ $[coin_0 := coin]([T_1]\langle coin = Head\rangle)$ sequential substitutions

  $\equiv$ $[T_1]\langle coin = Head\rangle$ no $coin_0$ in $[T_1]\langle coin = Head\rangle$

  $\equiv$ probabilistic choice substitution

  $$\tfrac{1}{2} \times [coin := Head]\langle coin = Head\rangle \;+\; \tfrac{1}{2} \times [\mathsf{abort}]\langle coin = Head\rangle$$

  $\equiv$ $\tfrac{1}{2} \times \langle Head = Head\rangle \;+\; \tfrac{1}{2} \times 0$ simple and abort substitutions

$\equiv \quad \frac{1}{2} \times 1 + 0$ embedded predicate, arithmetic

$\equiv \quad \frac{1}{2}$. arithmetic

Hence we have

$$\frac{1}{2} \quad \Rrightarrow \quad [coin_0 := coin][T_1]\langle coin = Head\rangle \ ,$$

and by Thm. 5, we conclude that

$$Prog.\ (5.8) \quad \sqsubseteq \quad T_1 \ .$$

- $T_2 \quad \widehat{=} \quad coin := Head.$

  This program always assigns $Head$ to $coin$ and clearly the probability that it establishes $coin = Head$ is 1. Consider

  $[coin_0 := coin][T_2]\langle coin = Head\rangle$

  $\equiv \quad [coin_0 := coin]([T_2]\langle coin = Head\rangle)$ sequential substitution

  $\equiv \quad [T_2]\langle coin = Head\rangle$ no $coin_0$ in $[T_2]\langle coin = Head\rangle$

  $\equiv \quad [coin := Head]\langle coin = Head\rangle$ definition of T_2

  $\equiv \quad \langle Head = Head\rangle$ simple substitution

  $\equiv \quad 1$. embedded predicate

  So, we have

  $$\frac{1}{2} \quad \Rrightarrow \quad [coin_0 := coin][T_2]\langle coin = Head\rangle \ ,$$

  and by Thm. 5, we conclude that

  $$Prog.\ (5.8) \quad \sqsubseteq \quad T_2 \ .$$

5.4 Probabilistic Loops

In Chap. 3, we gave a small extension to the proof obligation rule for loops to cover almost-certain termination. This extension is for the total correctness of loops, which ensures that the reasoning about probability-one termination with loops remains within the Boolean domain. We now turn to more general reasoning about loops in *pB*, i.e. we are concerned with partial correctness of probabilistic loops.

We will first recall the proof obligation rule for partial correctness of standard loops; then we set out the corresponding generalised proof obligation rule for probabilistic loops.

5.4.1 Proof Obligations for Standard Loops

For a standard loop, denoted *loop*, such as [3]

WHILE G **DO**
  S
INVARIANT I
END ,

we recall the proof obligations for its partial correctness, i.e. we try to reason about $[loop]\ Q$ where Q is a predicate representing the desired post-condition. The following conditions guarantee the partial correctness of the above loop [19]:

S1: The invariant I is maintained during the execution of the loop, i.e.

$$G \wedge I \quad \Rightarrow \quad [S]\, I \ .$$

S2: On termination, the post-condition Q is established, i.e.

$$\neg G \wedge I \quad \Rightarrow \quad Q \ .$$

5.4.2 Proof Obligations for Probabilistic Loops

We concentrate on giving the generalised version of the proof obligation rule for probabilistic loops. In general, a probabilistic loop will be calculated against a post-expectation B instead of a post-condition Q.

For a probabilistic loop, denoted *loop*, such as

[3] Here, we ignore the effect of the variant.

WHILE G **DO**
S
EXPECTATIONS E
END ,

we formulate the proof obligations for its partial correctness, i.e. we try to reason about $[loop]\, B$ where B is an real-value expression of the state represents the desire post-expectation. The following conditions guarantee the partial correctness of the above probabilistic loop:

P1: The expectation E is "maintained" (does not decrease) during the execution of the loop, i.e.

$$\langle G \rangle \times E \quad \Rrightarrow \quad [S]\, E\ .$$

P2: On termination, the post-expectation B is "established" (everywhere no less than the expectation E of the loop), i.e.

$$\langle \neg G \rangle \times E \quad \Rrightarrow \quad B\ .$$

The full semantics for probabilistic loops can be given with fixed-point notions and can be found elsewhere [49].

However, in practice, we usually calculate the pre-expectation of a probabilistic substitution with respect to a post-expectation of a particular form. This kind of post-expectation is usually a product of an embedded predicate [4]($\langle\cdot\rangle$) and another (general) expectation. The normal (standard) invariant is captured by the embedded predicate and the rest deals with the quantitative properties of the loop.

For practical reasons, we want to be able to separate the standard and probabilistic reasoning. The following lemma is provided for this purpose.

Lemma 8 *Let S be a probabilistic substitution; let P and Q be predicates; and let A and B be expectations. If we have*

$$\langle P \rangle \quad \Rrightarrow \quad [S]\langle Q \rangle\ ,\ \textit{and} \tag{5.13}$$

$$\langle P \rangle \times A \quad \Rrightarrow \quad [S]B\ ,\ \textit{then we have} \tag{5.14}$$

$$\langle P \rangle \times A \quad \Rrightarrow \quad [S](\langle Q \rangle \times B) \tag{5.15}$$

[4] Recall that an embedded predicate $\langle P \rangle$ is 1 if P holds and 0 otherwise.

Proof. In order to prove the lemma, we need to use the concept of "probabilistic conjunction operator". The operator (denoted $\&$) is defined over the expectation space as follows:

$$A \,\&\, B \quad \widehat{=} \quad (A + B - 1) \,\mathsf{max}\, 0 \,,$$

for any pair of expectations A and B. This operator is the probabilistic generalisation of the standard conjunction operation ($\wedge$) in Boolean domain. The properties of $\&$ operator can be found elsewhere [44, 49]. Here, we only need to use the following properties of the probabilistic conjunction operator:

Monotonic: For expectations A_1, A_2, B_1, B_2, if

$$A_1 \Rrightarrow B_1 \quad \text{and} \quad A_2 \Rrightarrow B_2 \,,$$

then we have

$$A_1 \,\&\, A_2 \quad \Rrightarrow \quad B_1 \,\&\, B_2 \,.$$

Sub-conjunctivity: For any pair of expectations A, B and probabilistic program S we have:

$$[S](A \,\&\, B) \quad \Lleftarrow \quad [S]A \,\&\, [S]B \,. \tag{5.16}$$

Special property related to embedded predicates: For any expectation E is in the range $[0..1]$, we have that

$$\langle P \rangle \times E \quad \equiv \quad \langle P \rangle \,\&\, E \,. \tag{5.17}$$

Returning to our lemma, we begin the proof with the left-hand side (here, we use the assumption that the expectations in this chapters are in the range $[0..1]$):

$\langle P \rangle \times A$

$\equiv$ $\langle P \rangle \times (\langle P \rangle \times A)$ arithmetic of $\langle \cdot \rangle$

$\equiv$ $\langle P \rangle \,\&\, (\langle P \rangle \times A)$ property (5.17) (since $0 \leq \langle P \rangle \times A \leq 1$)

$\Rrightarrow$ assumption (5.13), assumption (5.14) and monotonicity of $\&$

$[S]\langle Q \rangle \,\&\, [S]B$

$$\Rrightarrow \quad [S](\langle Q\rangle \,\&\, B) \qquad \text{sub-conjunctivity (5.16)}$$

$$\equiv \qquad \text{property (5.17) (since } B \text{ is within the range } [0..1]$$

$$[S](\langle Q\rangle \times B)\ ,$$

which completes the proof.

We can now use Lem. 8 to obtain the generalisation of the proof-obligation rule for the partial correctness of probabilistic loops. The total-correctness property can be proved using standard variant technique (in which, probabilistic choice substitutions are interpreted demonically), or the loop can be proved to terminate with probability-one using our technique from Chap. 3. We only consider partial correctness here.

Assuming that we have a loop (denoted as "*loop*") written in *pGSL* as follows:

WHILE G **DO**
  S
INVARIANT I
EXPECTATIONS E
END .

The partial correctness of the above loop (with respect to the post-expectation $\langle Q\rangle \times B$) is guaranteed by the following condition:

P1: The loop body cannot decrease the expected value of E if the invariant I and the guard G hold:

$$\langle G \wedge I\rangle \times E \quad \Rrightarrow \quad [S]\,(\langle I\rangle \times E)\ .$$

According to Lem. 8, this is achieved by the following two proof obligations:

P1a: The invariant I must hold within the loop body with probabilistic choice substitution being treated demonically — this is called *demonic retraction*. If $[\![S]\!]$ represents the demonic retraction of S [5], then this rule can

[5] This demonic retraction was introduced in Chap. 3 as

$$[\![S]\!] P \quad \equiv \quad ([S]\langle P\rangle = 1)\ .$$

This is defined to take advantage of the fact that $[S]\langle P\rangle$ can only take values in $\{0, 1\}$, and can be easily calculated by replacing all probabilistic choice substitutions by non-deterministic ones (provided that S contains no loops).

be formulated by

$$\langle G \wedge I \rangle \;\Rrightarrow\; [\![S]\!] \langle I \rangle,$$

or equivalently

$$G \wedge I \quad \Rightarrow \quad [\![S]\!] I \; .$$

P1b: The expectation E must not decrease within the loop body, i.e. the operation within the loop body cannot decrease the expectation E if the invariant I and the guard G hold:

$$\langle G \wedge I \rangle \times E \quad \Rrightarrow \quad [S]\, E \; .$$

P2: On termination, the loop establishes the post-expectation B with post-condition Q, i.e.

$$\langle \neg G \wedge I \rangle \times E \quad \Rrightarrow \quad \langle Q \rangle \times B \; .$$

According to Lem. 8 this can be achieved by the following:

P2a: On termination, the loop establishes the post-condition Q, that is we have

$$\langle \neg G \wedge I \rangle \;\Rrightarrow\; \langle Q \rangle \; .$$

We can rewrite this without embedding as

$$\neg G \wedge I \quad \Rightarrow \quad Q \; .$$

P2b: On termination, the loop establishes the post-expectation B:

$$\langle \neg G \wedge I \rangle \times E \quad \Rrightarrow \quad B \; .$$

If the above conditions are satisfied, and the total correctness condition is met, we have proved that

$$\langle I \rangle \times E \quad \Rrightarrow \quad [\mathit{loop}]\, (\langle Q \rangle \times B) \; .$$

5.5 Actual Changes to the *B-Toolkit*

The introduction of probabilistic specification substitution requires changes to be made to the *B-Toolkit* in order to support the new construct. We need to be able to write in some syntax the probabilistic specification substitution, namely, the

pre-expectation, the post-expectation and the frame (list of variables) of the substitution.

In standard *AMN*, the standard specification specification "$P \mid v : Q$" is written in the following form:

PRE P **THEN**
  ANY v' **WHERE** $[x_0, v := x, v']\, Q$ **THEN**
    $v := v'$
  END
END ,

where v' is a list of primed variables corresponding to v and assumed to be fresh, $[x_0, v := x, v']\, Q$ is the predicate in which x_0 is replaced by x and v is replaced by v' in Q. The standard specification is expressed in terms of a non-deterministic substitution with a precondition.

We can use the similar syntax for expressing the probabilistic specification substitution "$A \mid v : B$". Here, we consider a special kind of expectation which is a product of a (standard) embedded predicate and a (general) expectation (with the assumption that it is in the range $[0..1]$). In other words, we consider specifications of the form

$$(\langle P \rangle \times A) \mid v : (\langle Q \rangle \times B) \,.$$

The following code is equivalent to the above specification:

PRE $P \wedge expectation(A)$ **THEN**
  ANY v' **WHERE**
    $[x_0, v := x, v']\, Q \wedge expectation([x_0, v := x, v']\, B)$
  THEN
    $v := v'$
  END
END ,

in which the $expectation(\cdot)$ clause is just to distinguish between the embedded predicate and the expectation.

For the analyser, the toolkit will treat the above clause just as a normal substitution. There are some changes that need to be made to the type checker in order to verify that $expectation(A)$ and $expectation(B)$ are well-typed, i.e., to check that A and B are real numbers. Then, the type checker can treat $expectation(\cdot)$ as a predicate for well-typing. In other words, $expectation(\cdot)$ acts as a type casting from real to Boolean in the type-checking process.

In the proof-obligation generation phase, we need to make some changes in order to have correct obligations for maintaining the correctness of the refinement relation. Here, we separate the treatment of standard (predicates) from probabilistic (expectations) components. The obligations are generated according to Thm. 5 in Sec. 5.3.2.

Moreover, since the implementation can contain loops, the proof obligation generation process needs to be changed for loops at the same time. Again, we separate the obligations for standard and probabilistic components. That means extra proof obligations are generated according to rules *P1b*, *P2b* in Sec. 5.4.2. Obligations *P1a*, *P2a* are already generated as in the original toolkit.

5.6 The Min-Cut Algorithm

We use the example of finding a minimum cut of a graph to illustrate the application of the theorems in Sec. 5.3 and Sec. 5.4 in practice.

The Min-Cut algorithm operates on undirected and connected graphs. Between two connected nodes, there can be more than one edge. A *cut* is a set of edges such that if we remove just those edges, the graph will become disconnected. A *minimum* cut is a cut with the least number of edges. With this definition, a particular graph can have more than one minimum cut. The number of edges for a minimum cut is called the connectivity of the graph.

The algorithm has two parts. In the first part, a minimum cut of the graph is found probabilistically, but at low probability. In the second part, we use the well known technique of "probabilistic amplification" in order to increase the probability of finding a true minimum cut.

We will first briefly describe the Min-Cut algorithm and the probabilistic amplification technique; then we discuss how to code the Min-Cut algorithm in *pB*, and we look in particular at the proof obligations required.

5.6.1 Informal Description of the Min-Cut Algorithm: Contraction

Deterministic algorithms' complexities are often improved by randomisation, and Min-Cut algorithm is an example of that. The result for randomised algorithms is much better (in terms of time and complexity) than for the deterministic one,

especially for dense graphs [53].

The algorithm consists of a number of "contraction" steps. In a *contraction*, two connected nodes are chosen randomly and merged together. The resulting contracted graph then has one node fewer than the original one. Furthermore, the connectivity of the contracted graph is always no less than the original graph. It can be proved that any specific minimum cut in the original graph remains in the contraction with probability at least $\frac{NN-2}{NN}$ (where NN is the number of nodes in the graph). This step is done repeatedly until there are only two nodes left in the graph. Then the only cut left is the (multiple) edge connecting the last two nodes, and this will be the chosen cut. This cut is not guaranteed to be a minimum cut, but the number of edges for that cut cannot be less than connectivity of the graph (the number of edges in a minimum cut). If we are lucky, it will be in fact equal to the connectivity of the graph; and that is the point of probabilistic amplification, as we will see.

By "lucky" we mean that although the above contraction procedure does not guarantee to find a minimum cut for the original graph, there is a non-zero lower bound of probability that it will. It is easy to see that by multiplying the probabilities for the successive stages, the overall probability of finding a true minimum cut is at least

$$p(NN) \quad = \quad \frac{NN-2}{NN} \times \frac{NN-3}{NN-1} \times \cdots \times \frac{2}{4} \times \frac{1}{3} \quad = \quad \frac{2}{NN \times (NN-1)} \,. \tag{5.18}$$

This probability is relatively small, however, especially when there are a large number of nodes in the graph. Fortunately, further independent repetitions of the process can reduce the probability that a witness (solution to the problem) is not found on any of the repetitions. This is the *probabilistic amplification* technique that we describe below.

Full details of this algorithm are given by Motwani and Raghavan [53].

5.6.2 Probabilistic Amplification

When the search space contains a large number of possible solutions, it is difficult to find the right one deterministically. It often suffices however to choose an element at random from the search space, because there will be a small non-zero probability that the chosen element is a solution. This probability can be improved

by further independent repetitions of the process. This improvement is known as *probabilistic amplification*.

As an example, if we calculate the probability of finding a true minimum cut in one contraction test when $NN = 10$, it would be $\frac{2}{10\times 9}$, that is approximately 2%. We repeat the process of finding minimum cut in order to increase the probability. This can be shown by calculating the probability that a minimum cut is not found in any one of those tests. For MM tests, this probability will be at most

$$(1 - p(NN))^{MM} ,$$

since for each test, the probability of not finding a true minimum cut is at most $1 - p(NN)$. Hence, the overall probability of finding a true minimum cut for MM tests is at least

$$P(NN, MM) = 1 - (1 - p(NN))^{MM} ,$$

where $p(NN)$ is as in previous section.

For example, if we run the $NN = 10$ case 120 times, the error probability would only be around 10%, that is, our probability of success is increased from 2% to $100 - 10 = 90\%$.

5.6.3 Formal Development of Contraction

We will show how the contraction steps are specified and then subsequently implemented in *pAMN*; and we take a look at the proof obligations for preserving the refinement relationship between the specification and the implementation.

Specification of Contraction

We look at the specification of the contraction, i.e. of the one test to find the minimum cut. We take the abstract view for the specification since the probability that we are interested in depends only on the number of nodes in the original graph. The machine (specification) has no variables and has one operation to model the result of a single contraction. The input NN of the operation represents the number of nodes in the original graph. The output ans is of $BOOL$ type. It is $TRUE$ when we have found a true minimum cut, and $FALSE$ otherwise. In this specification, we want to state that for any input NN, the probability that the output ans is $TRUE$ on termination is at least $\frac{2}{NN\times(NN-1)}$ (as at (5.18)). The specification

MACHINE *contraction*
SEES *Bool_TYPE* , *Real_TYPE* , *Math*

OPERATIONS
ans ⟵ **contraction** (*NN*) ≙
PRE
$NN \in \mathbb{N} \wedge 2 \leq NN \wedge$
expectation (*frac* (2 , $NN \times (NN - 1)$))
THEN
ANY *aa* **WHERE**
$aa \in BOOL \wedge$ *expectation* (*emb* (*aa*))
THEN
ans := *aa*
END
END
END

Figure 5.1: Specification of contraction in *pAMN*

is shown in Fig. 5.1. The *pAMN* notation $emb(a)$ is equivalent to $\langle a \rangle$, where $a \in BOOL$ with the definition that $\langle TRUE \rangle = 1$ and $\langle FALSE \rangle = 0$.

An Implementation of Contraction

A loop implementation of the contraction is given in Fig. 5.2 on the next page. The *pAMN* notation $embedded(Q)$ is equivalent to $\langle Q \rangle$, where Q is a predicate.

In this implementation, we have a local variable nn to keep the number of nodes in the current graph, and so we start with $nn = NN$ (original graph) and $ans = TRUE$ (for the fact that there will be some minimum cuts in the original graph). At each stage, variable ans is $TRUE$ just when all actual minimum cuts have not yet been destroyed by any merge so far. We keep merging while the number of nodes is greater than 2. The operation *merge(nn, ans)* is specified in

IMPLEMENTATION *contractionI*
REFINES *contraction*
SEES *Bool_TYPE* , *Real_TYPE* , *merge* , *Math*

OPERATIONS
ans ⟵ **contraction** (*NN*) $\widehat{=}$
VAR *nn* **IN**
nn := *NN* **;** *ans* := *TRUE* **;**
WHILE $2 < nn$ **DO**
ans ⟵ *merge* (*nn* , *ans*) **;**
$nn := nn - 1$
VARIANT
nn
INVARIANT
$nn \in \mathbb{N} \wedge nn \leq NN \wedge 2 \leq nn \wedge ans \in BOOL \wedge$
expectation (*frac* (2 , $nn \times (nn - 1)$)) $\times$ *emb* (*ans*))
END
END
END

Figure 5.2: Implementation of contraction in *pAMN*

the machine merge as in Fig. 5.3 on the following page. The operation says that with probability "at most" $\frac{2}{nn}$, all minimum cuts will be destroyed by the contraction. Otherwise, there is a minimum cut that has not been destroyed. The merge operation can be explained by its equivalent *GSL* as follows:

$$\begin{array}{ll} & (ans := FALSE \ {}_{\frac{2}{nn}}\!\oplus\ ans := aa) \ [\!] \ ans := aa \\ \equiv & ans := FALSE \ {}_{\leq\frac{2}{nn}}\!\oplus\ ans := aa) \end{array}$$

MACHINE *merge*
SEES *Bool_TYPE* , *Real_TYPE*

OPERATIONS
ans ⟵ **merge** (*nn* , *aa*) $\widehat{=}$
PRE $nn \in \mathbb{N} \land aa \in BOOL$ **THEN**
CHOICE
PCHOICE *frac* (*2* , *nn*) **OF**
ans := *FALSE*
OR
ans := *aa*
END
OR
ans := *aa*
END
END
END

Figure 5.3: Specification of merge operation in *pAMN*

Proof Obligations of Contraction

We now will apply the generalised proof obligation rule for the probabilistic loop to prove the correctness of the implementation of the contraction process.

To prove the refinement relationship between the specification and its implementation of the contraction process, Thm. 5 is applied to those programs. The *pGSL* equivalent of the specification is

$$\langle NN \in \mathbb{N} \land 2 \leq NN \rangle \times p(NN) \mid ans : \langle ans \rangle \,.$$

We thus have to prove that

$$\begin{array}{ll} & \langle NN \in \mathbb{N} \land 2 \leq NN \rangle \times p(NN) \\ \Rrightarrow & [ans_0 := ans][contractionImpl]\langle ans \rangle \,, \end{array}$$

which can be simplified to

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \quad \Rrightarrow \quad [contractionImp]\langle ans \rangle \ ,$$

where we have used the fact that there are no ans_0 on the right-hand side, so that the substitution $[ans_0 := \ ans]$ is redundant.

We first state all the components of the loop (denoted $loop_1$) within our reasoning context. Referring to Sec. 5.4.2, we have the following information.

- The standard invariant is

$$I_1 \quad \widehat{=} \quad nn \in \mathbb{N} \ \wedge \ nn \leq NN \ \wedge \ 2 \leq nn \ \wedge \ ans \in BOOL \ .$$

- The guard for the loop is $G_1 \ \widehat{=} \ 2 < nn$.

- The body of the loop is

$$S_1 \ \widehat{=} \ ans \longleftarrow merge(nn, ans); \ nn := \ nn - 1 \ .$$

- The expectation (probabilistic invariant) is $E_1 \ \widehat{=} \ \frac{2}{nn \times (nn-1)} \times \langle ans \rangle$.

- The post-condition Q_1 is the constant predicate $true$.

- The post-expectation is $B_1 \ \widehat{=} \ \langle ans \rangle$.

Proving the partial correctness of $loop_1$ using the rules in Sec. 5.4.2 allows us to prove $\langle I_1 \rangle \times E_1$ instead of the $[loop_1] \langle ans \rangle$ (the total correctness is trivial with the declared variant's certain decrease for every iteration of the loop). So we have to prove that the (probabilistic) invariant is established initially, i.e.

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \Rrightarrow [nn := \ NN; ans := \ TRUE](\langle I_1 \rangle \times E_1) \ .$$

According to Lem. 8, it is equivalent to prove

$$NN \in \mathbb{N} \wedge 2 \leq NN \ \Rightarrow \ [nn := \ NN; ans := \ TRUE] I_1 \ ,$$

and under the assumption that $NN \in \mathbb{N} \wedge 2 \leq NN$, to prove

$$p(NN) \Rrightarrow [nn := \ NN; ans := \ TRUE] E_1 \ .$$

From all this, we have in fact that there are 14 proof obligations for the implementation of the contraction, all of which have been proved using the modified *B-Toolkit* with some extra proof rules.

Proving the Obligations

Here, we look at the proof for the maintenance of E_1 during the execution of $loop_1$ (proof obligation *P1b* in Sec. 5.4.2). We have to prove that

$$\langle G_1 \wedge I_1 \rangle \times E_1 \;\Rrightarrow\; [S_1]E_1 \;.$$

The complete proof of the development can be found in [24]. Furthermore, we have used the fact that proving

$$\langle P \rangle \times A \Rrightarrow B \tag{5.19}$$

is equivalent to proving

$$A \Rrightarrow B$$

under the assumption that P holds [6]— so it is sufficient to prove

$$E_1 \;\Rrightarrow\; [S_1]E_1 \;,$$

under the assumption $G_1 \wedge I_1$. We start from the right-hand side as follows:

$$[S_1]E_1$$

$\equiv$ definition of S_1 and E_1

$$[ans \longleftarrow merge(nn, ans);\; nn := nn - 1] \left(\tfrac{2}{nn \times (nn-1)} \times \langle ans \rangle \right)$$

$\equiv$ sequential substitution

$$[ans \longleftarrow merge(nn, ans)]\,[nn := nn - 1] \left(\tfrac{2}{nn \times (nn-1)} \times \langle ans \rangle \right)$$

$\equiv$ simple substitution

$$[ans \longleftarrow merge(nn, ans)] \left(\tfrac{2}{(nn-1) \times (nn-2)} \times \langle ans \rangle \right)$$

$\equiv$ definition of merge

$$\left[\begin{array}{ll} & (ans := FALSE \;\; {}^{2}\!/_{nn} \oplus \; ans := ans) \\ [\!] & ans := ans \end{array} \right] \left(\tfrac{2}{(nn-1) \times (nn-2)} \times \langle ans \rangle \right)$$

[6] When P does not hold, the left-hand side of (5.19) is zero, which makes the inequality holds trivially.

$\equiv$ non-deterministic substitution

$$\min \begin{array}{l} \left[ans := \ FALSE \ {}_{\frac{2}{nn}}\oplus \ ans := \ ans\right]\left(\frac{2}{(nn-1)\times(nn-2)} \times \langle ans\rangle\right) \\ [ans := \ ans]\left(\frac{2}{(nn-1)\times(nn-2)} \times \langle ans\rangle\right) \end{array}$$

$\equiv$ probabilistic and simple substitution

$$\min \begin{array}{l} \left(\begin{array}{lll} \frac{2}{nn} & \times & \left(\frac{2}{(nn-1)\times(nn-2)} \times \langle FALSE\rangle\right) \\ + \ \left(1-\frac{2}{nn}\right) & \times & \left(\frac{2}{(nn-1)\times(nn-2)} \times \langle ans\rangle\right) \end{array}\right) \\ \left(\frac{2}{(nn-1)\times(nn-2)} \times \langle ans\rangle\right) \end{array}$$

$\equiv$ embedded predicate and arithmetic

$$\min \begin{array}{l} \left(\frac{2}{nn} \times \left(\frac{2}{(nn-1)\times(nn-2)} \times 0\right) + \frac{nn-2}{nn} \times \frac{2}{(nn-1)\times(nn-2)} \times \langle ans\rangle\right) \\ \left(\frac{2}{(nn-1)\times(nn-2)} \times \langle ans\rangle\right) \end{array}$$

$\equiv$ arithmetic

$$\left(\frac{2}{nn\times(nn-1)} \times \langle ans\rangle\right) \ \min \ \left(\frac{2}{(nn-1)\times(nn-2)} \times \langle ans\rangle\right)$$

$\equiv$ $\frac{2}{nn\times(nn-1)} \times \langle ans\rangle$ $\frac{2}{nn\times(nn-1)} < \frac{2}{(nn-1)\times(nn-2)}$ when $2 \leq nn$

$\equiv$ E_1 . definition of E_1

So we have proved that $E_1 \ \Rrightarrow \ [S_1]E_1$. Proofs of the other obligations can be found in [24].

5.6.4 Formal Development of Probabilistic Amplification

This section discusses the use of the probabilistic amplification technique in order to increase the probability of finding a true minimum cut. We start by looking at the specification and its implementation using a probabilistic loop, and then we look at the proof obligations for the refinement step. We will show that a

slightly more specialised version of the probabilistic specification substitution (and the corresponding fundamental theorem) is required for developments of this kind.

Specification of Min-Cut Probabilistic Amplification

Similar to specification of the contraction machine, the specification MinCut does not have any variables to represent its state. The machine has only one operation, namely minCut. As well as the input NN representing the number of nodes in the original graph, the operation has one extra input MM that represents the number of times we "amplify" the probability, i.e. the number of times we will carry out a contraction. The output ans of the operation again abstractly models whether we find a true minimum cut or not after the amplification process. We begin with a "one-shot" specification in Fig. 5.4 on the next page, similar to the contraction.

The specification states that the probability of finding the correct minimum cut should be at least

$$P(NN, MM) \;=\; 1 - (1 - p(NN))^{MM} \;,$$

where $p(NN) = \frac{2}{NN \times (NN-1)}$.

Implementation of Min-Cut Probabilistic Amplification

The implementation of the probabilistic amplification is shown in Fig. 5.5. In the implementation, we use two auxiliary variables, namely mm and aa, which represent the counter and the recent output from the contraction process, respectively. In the beginning, mm is assigned MM because we want to repeat the contraction process MM times; and ans is assigned $FALSE$ since we have not found the right minimum cut yet. In the body of the loop, a contraction process is taken and its result is carried out in aa; then ans is the disjunction of the new result aa and the old ans (since if we find the correct (least) cut once, we can never lose it); and finally, the counter decreases accordingly.

Proof Obligations of Min-Cut Probabilistic Amplification

We apply Thm. 5 for refining a probabilistic specification substitution and the rules in Sec. 5.4.2 for proof obligations for partial correctness of loops in order to prove the correctness of the implementation. The total correctness of the loop in this case

MACHINE *minCut*
SEES
Bool_TYPE , *Real_TYPE* , *Math*

OPERATIONS
ans ⟵ **minCut** (*NN* , *MM*) ≙
PRE
$NN \in \mathbb{N} \wedge 2 \leq NN \wedge MM \in \mathbb{N}_1 \wedge$
expectation (*real* (*1*) −
power (*real* (*1*) − *frac* (*2* , *NN* × (*NN* − *1*)) , *MM*))
THEN
ANY *aa* **WHERE**
$aa \in BOOL \wedge$ *expectation* (*emb* (*aa*))
THEN
ans := *aa*
END
END
END

Figure 5.4: Specification of MinCut probabilistic amplification in *pAMN*

is trivial since the variant mm decreases for every iteration of the loop. The *pGSL* equivalent of the probabilistic amplification specification is:

$$\langle NN \in \mathbb{N} \ \wedge \ 2 \leq NN \ \wedge \ MM \in \mathbb{N}_1 \rangle \times P(NN, MM) \ \mid \ ans : \langle ans \rangle \ .$$

Applying Thm. 5, we must show that

$$\begin{array}{ll} & \langle NN \in \mathbb{N} \ \wedge \ 2 \leq NN \ \wedge \ MM \in \mathbb{N}_1 \rangle \times P(NN, MM) \\ \Rrightarrow & [ans_0 := \ ans][minCutImplementation]\langle ans \rangle \ , \end{array} \tag{5.20}$$

in order to establish the refinement. The implication (5.20) can be simplified to

$$\begin{array}{ll} & \langle NN \in \mathbb{N} \ \wedge \ 2 \leq NN \ \wedge \ MM \in \mathbb{N}_1 \rangle \times P(NN, MM) \\ \Rrightarrow & [minCutImplementation]\langle ans \rangle \ , \end{array}$$

IMPLEMENTATION *minCutI*
REFINES *minCut*
SEES *Bool_TYPE* , *Real_TYPE* , *Math* , *contraction* , *Bool_TYPE_Ops*

OPERATIONS
ans ⟵ **minCut** (*NN* , *MM*) $\widehat{=}$
VAR *mm* , *aa* **IN**
mm := *MM* **;** *ans* := *FALSE* **;**
WHILE *mm* > *0* **DO**
aa ⟵ *contraction* (*NN*) **;**
ans ⟵ *DIS_BOOL* (*ans* , *aa*) **;**
mm := *mm* − *1*
VARIANT
mm
INVARIANT
$mm \in \mathbb{N} \wedge mm \leq MM \wedge ans \in BOOL \wedge$
expectation (*emb* (*ans*) + *embedded* (¬ (*mm* = *0*)) × *emb* (*neg* (*ans*)) ×
(*real* (*1*) − *power* (*real* (*1*) − *frac* (*2* , *NN* × (*NN* − *1*)) , *mm*)))
END
END
END

Figure 5.5: Implementation of MinCut probabilistic amplification in *pAMN*

by noting that there are no occurrences of ans_0 on the right-hand side. Also, we again separate the standard predicates and the expectations, i.e. we will prove

$$P(NN, MM) \quad \Rrightarrow \quad [minCutImplementation]\langle ans \rangle \,,$$

under the assumption that $NN \in \mathbb{N} \ \wedge \ 2 \leq NN \ \wedge \ MM \in \mathbb{N}_1$.

Referring to Sec. 5.4.2, we obtain the following information about all the components of the loop (denoted $loop_2$) within our reasoning context:

- The standard invariant is

$$I_2 \;\widehat{=}\; mm \in \mathbb{N} \;\wedge\; mm \leq MM \;\wedge\; ans \in BOOL\,.$$

- The guard for the loop is $G_2 \mathrel{\widehat{=}} mm \neq 0$.

- The body of the loop is

$$S_2 \;\widehat{=}\; aa \longleftarrow contraction(NN);\; ans := ans \vee aa;\; mm := mm - 1\,.$$

- The expectation (probabilistic invariant) is

$$E_2 \;\widehat{=}\; \langle ans \rangle \;+\; \langle mm \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm)\,.$$

- The post-condition Q_2 is the constant predicate $true$.

- The post-expectation is $B_2 \mathrel{\widehat{=}} \langle ans \rangle$.

Proving the correctness of $loop_2$ using the rules in Sec. 5.4.2 allows us to prove $\langle I_2 \rangle \times E_2$ instead of $[loop_2]\,\langle ans \rangle$. So we have to prove that

$$\begin{array}{ll} & \langle NN \in \mathbb{N} \;\wedge\; 2 \leq NN \;\wedge\; MM \in \mathbb{N}_1 \rangle \times P(NN, MM) \\ \Rrightarrow & [mm := MM; ans := FALSE](\langle I_2 \rangle \times E_2)\,. \end{array}$$

According to Lem. 8, it is equivalent to prove

$$NN \in \mathbb{N} \;\wedge\; 2 \leq NN \;\wedge\; MM \in \mathbb{N}_1 \;\Rightarrow\; [mm := MM; ans := FALSE] I_2\,,$$

and under the assumption that $NN \in \mathbb{N} \;\wedge\; 2 \leq NN \;\wedge\; MM \in \mathbb{N}_1$, to prove

$$P(NN, MM) \Rrightarrow [mm := MM; ans := FALSE] E_2\,.$$

From the above information there are 14 proof obligations for the implementation of the contraction, 13 of which have been proved using the modified *B-Toolkit* with some extra proof rules. We will look at the remaining proof obligation in the next section.

An Undischargeable Obligation

Here, we concentrate only on the proving of *P1b* in Sec. 5.4.2, i.e. the maintenance of E_2 during the execution of the loop. The complete proofs of other obligations can be seen elsewhere [24]. For *P1b*, we have to prove that

$$\langle G_2 \wedge I_2 \rangle \times E_2 \Rrightarrow [S_2]\, E_2 \ ,$$

which is equivalent to proving

$$E_2 \Rrightarrow [S_2]\, E_2 \ ,$$

under the assumption that G_2 and I_2 both hold. We begin the proof from the right-hand side.

$$[S_2]\, E_2$$

$\equiv$ definition of S_2 and E_2

$$\left[\begin{array}{l} aa \longleftarrow contraction(NN); \\ ans := ans \vee aa; \\ mm := mm - 1 \end{array}\right] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm) \end{array}\right)$$

$\equiv$ sequential substitution

$$\left[\begin{array}{l} aa \longleftarrow contraction(NN); \\ ans := ans \vee aa \end{array}\right] [mm := mm - 1] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm) \end{array}\right)$$

$\equiv$ simple substitution

$$\left[\begin{array}{l} aa \longleftarrow contraction(NN); \\ ans := ans \vee aa \end{array}\right] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm - 1) \end{array}\right)$$

$\equiv$ sequential substitution

$$[aa \longleftarrow contraction(NN)]\, [ans := ans \vee aa] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm - 1) \end{array}\right)$$

$\equiv$ simple substitution

$$[aa \longleftarrow contraction(NN)] \left(\begin{array}{l} \langle ans \vee aa \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg (ans \vee aa) \rangle \times P(NN, mm - 1) \end{array}\right)$$

$\equiv$ definition of contraction

$$[\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \mid aa : \langle aa \rangle] \left(\begin{array}{l} \langle ans \vee aa \rangle \; + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg (ans \vee aa) \rangle \times P(NN, mm - 1) \end{array} \right)$$

$\equiv$ probabilistic specification substitution

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \times \sqcap aa \cdot \left(\left(\begin{array}{l} \langle ans \vee aa \rangle \; + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg (ans \vee aa) \rangle \times P(NN, mm - 1) \end{array} \right) \div \langle aa \rangle \right)$$

$\equiv$ minimum when $aa = TRUE$, see below

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \times \left(\begin{array}{l} \langle ans \vee TRUE \rangle \; + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg (ans \vee TRUE) \rangle \times P(NN, mm - 1) \end{array} \right)$$

$\equiv$ logic

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \times \left(\begin{array}{l} \langle TRUE \rangle \; + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg TRUE \rangle \times P(NN, mm - 1) \end{array} \right)$$

$\equiv$ embedded predicate

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \times \left(\begin{array}{l} 1 \; + \\ \langle mm - 1 \neq 0 \rangle \times 0 \times P(NN, mm - 1) \end{array} \right)$$

$\equiv$ arithmetic

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \times 1$$

$\equiv$ arithmetic

$$\langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \; .$$

For the deferred judgement, when $aa = FALSE$, $\langle aa \rangle$ is zero which sets the value of fraction to infinity —and thus the minimum $\sqcap$ will ignore it.

So we have to prove that

$$\langle ans \rangle + \langle mm \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm) \Rrightarrow \langle NN \in \mathbb{N} \wedge 2 \leq NN \rangle \times p(NN) \; ,$$

which in fact does not hold. We found that this is a problem related to termination. This is not surprising in retrospect, because our definition of probabilistic specification substitution does not require that the specification always terminates.

For example, a specification "$p \mid v : \langle Q \rangle$" which is ensured to establish Q with probability at least p might still with probability $1 - p$ not establish Q, or even fail to terminate at all. In the example, we rely on the fact that even in the case contraction does not find a true minimum cut, it still terminates and returns aa is $FALSE$. Hence, we must ensure that the specification does terminate in any case, even though we do not care about which post-condition it establishes. In the next section, we will address the problem by introducing the terminating probabilistic specification substitution and its corresponding fundamental theorem.

5.7 Terminating Probabilistic Specification Substitutions

In order to avoid the problem revealed in the last section, we have to introduce the concept of "terminating probabilistic specification substitution" and a corresponding fundamental theorem for it. We also reconstruct the proof obligations for refinement of the new *terminating* probabilistic amplification program.

Here, we consider a special case of the probabilistic substitution, where the post-expectation B is standard, i.e is in the form $\langle Q \rangle$ for some predicate Q; and the pre-expectation A is the probability —still a function of the state— that Q will be achieved. For consistency with probability elsewhere, we use lower-case p for the pre-expectation.

The specification substitutions defined in Sec. 5 do not exclude aborting programs. In *B*, however, all programs are proved to terminate. So we want to take advantage of that and propose the *terminating probabilistic specification substitution*. It is defined as follows.

Definition 6 *Let p be a probabilistic expression over x which is free from x_0; and let Q be a predicate defined over x_0, v and satisfying*

$$\forall\, x_0 \cdot (\exists\, v \cdot Q)\ . \tag{5.21}$$

The specification $\{p \mid v : \langle Q \rangle\}$ is defined by:

$$\{p \mid v : \langle Q \rangle\} \quad \widehat{=} \quad (1 \mid v : \langle Q \rangle) \quad {}_p\!\oplus \quad (1 \mid x : 1)\ . \tag{5.22}$$

That is, the specification $\{p \mid v : \langle Q \rangle\}$ establishes Q with probability at least p; but even if it does not, it still terminates [7]. Here, termination is represented by the

[7] The termination is what distinguishes it from "$p \mid v : \langle Q \rangle$".

specification "$1 \mid x : 1$", corresponding to standard program "$\mathsf{true} \mid x : \mathsf{true}$".

Secondly, we introduce the fundamental theorem for the above terminating substitution as follows.

Theorem 6 *Let p be an expression over x; let Q be a predicate defined over x_0, v, and satisfying*

$$\forall\, x_0 \cdot (\exists\, v \cdot Q)\ ;$$

and let T be any program. For all such programs T, if

$$(1 \mid x : 1) \quad \sqsubseteq \quad T \text{ , i.e. if } T \text{ terminates}$$

and if

$$(p \mid v : \langle Q \rangle) \quad \sqsubseteq \quad T$$

then in fact

$$\{p \mid v : \langle Q \rangle\} \quad \sqsubseteq \quad T\ .$$

Proof. Let E be an arbitrary expectation over x. For any x_0, we have from arithmetic that

$$E \quad \Leftarrow \quad \sqcap\, x \cdot (E) + (\sqcap x \cdot (E \div \langle Q^w \rangle) - \sqcap x \cdot (E)) \times \langle Q^w \rangle \tag{5.23}$$

where

$$Q^w \ \widehat{=}\ Q \wedge w = w_0\ .$$

We have the above inequality because of the non-empty condition (5.21) of Q in Def. 6 and because the embedded predicate $\langle Q^w \rangle$ can have value only 1 or 0. And in each case, it can be easily proved that the inequality holds.

First, we estimate the pre-expectation of T with respect to E: for all x_0 we have

$$[T]E$$

$\Leftarrow$ (5.23) and monotonicity of T

$$[T]\,(\sqcap x \cdot (E) + (\sqcap x \cdot (E \div \langle Q^w \rangle) - \sqcap x \cdot (E)) \times \langle Q^w \rangle)$$

$\Leftarrow$ $\sqcap x \cdot (E) \times [T]1 +$ sublinearity of T, see below
$(\sqcap x \cdot (E \div \langle Q^w \rangle) - \sqcap x \cdot (E)) \times [T]\langle Q^w \rangle\ .$

For the deferred judgement, we use the sublinearity property [49] of probabilistic programs, which states that, for any non-negative real constants c_1, c_2, any post expectations B_1, B_2, and any program S, we have

$$[S]\,(c_1 \times B_1 \;+\; c_2 \times B_2) \quad \Leftarrow \quad c_1 \times [S]\,B_1 \;+\; c_2 \times [S]\,B_2\;. \tag{5.24}$$

Hence (since x_0 is unconstrained), in particular,

$$[T]E$$

$\Leftarrow$ x_0 is given a value x

$$[x_0 :=\, x]\left(\begin{array}{l} \sqcap x \cdot (E) \times [T]1 \;+ \\ (\sqcap x \cdot (E \;\div\; \langle Q^w\rangle) \;-\; \sqcap\, x \cdot (E)) \times [T]\langle Q^w\rangle \end{array}\right)$$

$\equiv$ simple substitution $x_0 :=\, x$

$$\begin{array}{l} [x_0 :=\, x](\sqcap x \cdot (E) \times [T]1) \;+ \\ ([x_0 :=\, x]\,(\sqcap x \cdot (E \;\div\; \langle Q^w\rangle) \;-\; \sqcap\, x \cdot (E))) \times [x_0 :=\, x][T]\langle Q^w\rangle \end{array}$$

$\equiv$ x_0 is a fresh variable and simple substitution

$$\begin{array}{l} \sqcap x \cdot (E) \times [x_0 :=\, x][T]1 \;+ \\ ([x_0 :=\, x] \;\sqcap\, x \cdot (E \;\div\; \langle Q^w\rangle) \;-\; \sqcap\, x \cdot (E)) \times [x_0 :=\, x][T]\langle Q^w\rangle \end{array}$$

$\Leftarrow$ $\sqcap x \cdot (E) \times 1+$ refinement assumption
$([x_0 :=\, x] \;\sqcap\, x \cdot (E \;\div\; \langle Q^w\rangle) - \sqcap x \cdot (E)) \times p$

$\equiv$ arithmetic

$$\sqcap x \cdot (E) \;\times\; (1-p) \quad + \quad [x_0 :=\, x] \;\sqcap\, x \cdot (E \;\div\; \langle Q^w\rangle) \;\times\; p$$

$\equiv$ $[1 \mid x : 1]E \;\times\; (1-p) \quad + \quad [1 \mid v : \langle Q\rangle]E \;\times\; p$ Def. 5

$\equiv$ $[(1 \mid v : \langle Q\rangle) \quad {}_p\!\oplus \quad (1 \mid x : 1)]E$ probabilistic choice substitution

$\equiv$ $[\{p \mid v : \langle Q\rangle\}]E\;.$ Def. 6

Because E is arbitrary, therefore

$$\{p \mid v : \langle Q\rangle\} \sqsubseteq T\;.$$

Since the proof obligations for termination are always generated by the *B-Toolkit*, from now on we will assume that we always use the terminating version of probabilistic specification substitutions.

With the new terminating version of the specification and fundamental theorem, we can reconstruct and prove all the proof obligations for the implementation of probabilistic amplification as follows.

Recall from Sec. 5.6.4 that we have to prove the maintenance of E_2 during the execution of the loop, i.e. we have to prove

$$E_2 \quad \Rrightarrow \quad [S_2]\, E_2 \,, \tag{5.25}$$

under the assumption that G_2 and I_2 both hold. Similar calculations can be carried out, where we just need to replace the specification of contraction with its terminating version. We begin the proof from the right-hand side:

$$[S_2]\, E_2$$

$\equiv$ definition of S_2 and E_2

$$\left[\begin{array}{l} aa \longleftarrow contraction(NN); \\ ans := ans \vee aa; \\ mm := mm - 1 \end{array}\right] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm) \end{array}\right)$$

$\equiv$ sequential substitution

$$\left[\begin{array}{l} aa \longleftarrow contraction(NN); \\ ans := ans \vee aa \end{array}\right] [mm := mm - 1] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm) \end{array}\right)$$

$\equiv$ simple substitution

$$\left[\begin{array}{l} aa \longleftarrow contraction(NN); \\ ans := ans \vee aa \end{array}\right] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm - 1) \end{array}\right)$$

$\equiv$ sequential substitution

$$[aa \longleftarrow contraction(NN)]\, [ans := ans \vee aa] \left(\begin{array}{l} \langle ans \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm - 1) \end{array}\right)$$

$\equiv$ simple substitution

$$[aa \longleftarrow contraction(NN)]$$
$$\Big(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\Big)$$

$\equiv$ definition of "terminating" contraction

$$[\{\langle NN \in \mathbb{N} \wedge 2 \leq NN\rangle \times p(NN) \mid aa : \langle aa\rangle\}]$$
$$\Big(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\Big)$$

$\equiv$ terminating probabilistic specification substitution

$$\big[(1 \mid aa : \langle aa\rangle) \quad {}_{\langle NN \in \mathbb{N} \wedge 2 \leq NN\rangle \times p(NN)}\oplus \quad (1 \mid aa : 1)\big]$$
$$\Big(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\Big)$$

$\equiv$ context information guarantees that $NN \in \mathbb{N} \wedge 2 \leq NN$

$$\big[(1 \mid aa : \langle ans\rangle) \quad {}_{p(NN)}\oplus \quad (1 \mid aa : 1)\big]$$
$$\Big(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\Big)$$

$\equiv$ probabilistic choice substitution

$$p(NN) \times [1 \mid aa : \langle aa\rangle]$$
$$\Big(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\Big) \; +$$
$$(1 - p(NN)) \times [1 \mid aa : 1]$$
$$\Big(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\Big)$$

$\equiv$ probabilistic specification substitution

$$p(NN) \; \times$$
$$\sqcap aa \cdot \left(\left(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\right) \div \langle aa\rangle\right) \; +$$
$$(1 - p(NN)) \; \times$$
$$\sqcap aa \cdot \left(\left(\langle ans \vee aa\rangle \; + \; \langle mm - 1 \neq 0\rangle \times \langle \neg(ans \vee aa)\rangle \times P(NN, mm-1)\right) \div 1\right)$$

$\equiv$ the first minimum established when $aa = TRUE$

$$p(NN) \times \begin{pmatrix} \langle ans \vee TRUE \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg(ans \vee TRUE) \rangle \times P(NN, mm-1) \end{pmatrix} +$$
$$(1 - p(NN)) \times$$
$$\sqcap aa \cdot \begin{pmatrix} \langle ans \vee aa \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg(ans \vee aa) \rangle \times P(NN, mm-1) \end{pmatrix}$$

$\equiv$ logic and the expand the minimum for $aa = TRUE$ or $aa = FALSE$

$$p(NN) \times \begin{pmatrix} \langle TRUE \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg TRUE \rangle \times P(NN, mm-1) \end{pmatrix} +$$
$$(1 - p(NN)) \times$$
$$\begin{pmatrix} \min & \begin{pmatrix} \langle ans \vee TRUE \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg(ans \vee TRUE) \rangle \times P(NN, mm-1) \end{pmatrix} \\ & \begin{pmatrix} \langle ans \vee FALSE \rangle \ + \\ \langle mm - 1 \neq 0 \rangle \times \langle \neg(ans \vee FALSE) \rangle \times P(NN, mm-1) \end{pmatrix} \end{pmatrix}$$

$\equiv$ logic and embedded predicate

$$p(NN) \times \begin{pmatrix} 1 \ + \\ \langle mm - 1 \neq 0 \rangle \times 0 \times P(NN, mm-1) \end{pmatrix} +$$
$$(1 - p(NN)) \times$$
$$\begin{pmatrix} \min & \big(1 \ + \ \langle mm - 1 \neq 0 \rangle \times 0 \times P(NN, mm-1)\big) \\ & \big(\langle ans \rangle \ + \ \langle mm - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm-1)\big) \end{pmatrix}$$

$\equiv$ arithmetic

$$p(NN) \times 1 +$$
$$(1 - p(NN)) \times$$
$$\big(1 \quad \min \quad \big(\langle ans \rangle \ + \ \langle mm - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm-1)\big)\big)$$

$\equiv$ the right-hand side of $\min$ is alway less than 1

$$p(NN) +$$
$$(1 - p(NN)) \times \big(\langle ans \rangle + \langle mm - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, mm-1)\big) \ .$$

Now we consider two cases: $mm = 1$ and $mm \neq 1$

- When $mm = 1$, the left-hand side of (5.25) (i.e. E_2) is the same as:

$$\langle ans \rangle \ + \ \langle \neg ans \rangle \times P(NN, 1) \quad \equiv \quad \langle ans \rangle \ + \ \langle \neg ans \rangle \times p(NN) \ .$$

The right-hand side of (5.25) is equivalent to

$$p(NN) + \\ (1 - p(NN)) \times \big(\langle ans \rangle + \langle 1 - 1 \neq 0 \rangle \times \langle \neg ans \rangle \times P(NN, 1 - 1)\big)$$

$\equiv$ logic and arithmetic

$$p(NN) + \\ (1 - p(NN)) \times \big(\langle ans \rangle + \langle FALSE \rangle \times \langle \neg ans \rangle \times P(NN, 1 - 1)\big)$$

$\equiv$ embedded predicate and arithmetic

$$p(NN) \quad + \quad (1 - p(NN)) \times \langle ans \rangle$$

$\equiv$

$$\langle ans \rangle \quad + \quad (1 - \langle ans \rangle) \times p(NN)$$

arithmetic

$\equiv$

$$\langle ans \rangle \quad + \quad \langle \neg ans \rangle \times p(NN)\,,$$

logic

which is the same as the right-hand side.

- When $mm \neq 1$, we simplify the right-hand side of (5.25) further as follows:

$$p(NN) + \\ (1 - p(NN)) \times \big(\langle ans \rangle + \langle TRUE \rangle \times \langle \neg ans \rangle \times P(NN, mm - 1)\big)$$

$\equiv$ embedded predicate

$$p(NN) + \\ (1 - p(NN)) \times (\langle ans \rangle + 1 \times \langle \neg ans \rangle \times P(NN, mm - 1))$$

$\equiv$ arithmetic

$$p(NN) + (1 - p(NN)) \times (\langle ans \rangle + \langle \neg ans \rangle \times P(NN, mm - 1))$$

$\equiv$ arithmetic and $\langle aa \rangle + \langle \neg aa \rangle = 1$

$$p(NN) \times (\langle ans \rangle + \langle \neg ans \rangle) \; + \\ (1 - p(NN)) \times \langle aa \rangle + (1 - p(NN)) \times P(NN, mm - 1) \times \langle \neg aa \rangle$$

$\equiv$ arithmetic

$$(p(NN) + (1 - p(NN))) \times \langle ans \rangle \; + \\ (p(NN) + (1 - p(NN)) \times P(NN, mm - 1)) \times \langle \neg ans \rangle$$

$\equiv$ arithmetic and definition of P(NN,mm- 1)

$$1 \times \langle ans \rangle \; + \\ (p(NN) + (1 - p(NN)) \times (1 - (1 - p(NN))^{mm-1})) \times \langle \neg ans \rangle$$

$\equiv$ arithmetic

$$\langle ans \rangle \; + \\ (p(NN) + (1 - p(NN)) - (1 - p(NN))^{mm}) \times \langle \neg ans \rangle$$

$\equiv$ $\langle ans \rangle \; + \; (1 - (1 - p(NN))^{mm}) \times \langle \neg ans \rangle$ arithmetic

$\equiv$ $\langle ans \rangle \; + \; P(NN, mm) \times \langle \neg ans \rangle \; .$ definition of P(NN, mm)

For the left-hand side of (5.25), we have the equivalent

$$\langle ans \rangle \; + \; \langle \neg ans \rangle \times P(NN, mm) \; ,$$

since the standard invariant guarantees that $mm \neq 0$. Thus the probabilistic invariant E_2 is indeed maintained by the loop.

5.8 Specification Frame

In this section, we discuss the frame of our probabilistic specification substitution and the intuitive meaning (somewhat surprising) related to it.

Consider the following program

$$\left\{ \frac{1}{2} \;\middle|\; coinA : \langle coinA = Head \rangle \right\} \; , \tag{5.26}$$

which guarantees to terminate and establish that $coinA$ is $Head$ with probability at least $\frac{1}{2}$. What about other variables in our system? Assume that the program state contains another coin, namely $coinB$. We consider the post-expectation where $coinB$ is $Head$. Applying the semantics from Sec. 5.7, we have

$$\left[\left\{\tfrac{1}{2} \mid coinA : \langle coinA = Head\rangle\right\}\right]\langle coinB = Head\rangle$$

$\equiv$ terminating probabilistic specification substitution

$$\left[(1 \mid coinA : \langle coinA = Head\rangle) \quad {}_{\frac{1}{2}}\!\oplus \quad (1 \mid (coinA, coinB) : 1)\right]\langle coinB = Head\rangle$$

$\equiv$ probabilistic choice substitution

$$\begin{array}{l} \tfrac{1}{2} \times [1 \mid coinA : \langle coinA = Head\rangle]\langle coinB = Head\rangle \\ + \\ \tfrac{1}{2} \times [1 \mid (coinA, coinB) : 1]\langle coinB = Head\rangle \end{array}$$

$\equiv$ probabilistic specification substitution

$$\begin{array}{l} \tfrac{1}{2} \times 1 \times [coinA_0, coinB_0 := coinA, coinB] \\ \sqcap coinA \cdot (\langle coinB = Head\rangle \div (\langle coinA = Head\rangle \times \langle coinB = coinB_0\rangle)) \\ + \\ \tfrac{1}{2} \times 1 \times [coinA_0, coinB_0 := coinA, coinB] \\ \sqcap (coinA, coinB) \cdot (\langle coinB = Head\rangle \div 1) \end{array}$$

$\equiv$ the first minimum is $\langle coinB_0 = Head\rangle$ when $coinA$ is $Head$ and $coinB$ is $coinB_0$; the second minimum is 0 when $coinB$ is $Tail$

$$\begin{array}{ll} & \tfrac{1}{2} \times [coinA_0, coinB_0 := coinA, coinB]\langle coinB_0 = Head\rangle \\ + & \tfrac{1}{2} \times [coinA_0, coinB_0 := coinA, coinB]\, 0 \end{array}$$

$\equiv$ $\tfrac{1}{2} \times \langle coinB = Head\rangle$. arithmetic and simple substitution

The semantics of the substitution tell us that it establishes the post-expectation $\langle coinB = Head\rangle$ with probability at least $\frac{1}{2} \times \langle coinB = Head\rangle$. If we start in the state where $coinB$ is indeed $Head$, the guaranteed probability we get is $\frac{1}{2}$. However, we might expect the probability to be 1 (not $\frac{1}{2}$) in the case where $coinB$ is $Head$ initially, since $coinB$ is not in the frame. Hence we cannot guarantee that the system keeps $coinB$ unchanged (always). In fact, we can only guarantee that $coinB$ is unchanged at least half of the time (when the program also establishes the desired post-expectation $\langle coinA = Head\rangle$). The reason for this apparent anomaly is the right-hand branch, executed with probability $\frac{1}{2}$, which can set $coinB$ arbitrary.

This result came as a surprise at first [8], but the semantics of the terminating probabilistic specification substitution clearly allows this behaviour. The condition for termination "$1 \mid x : 1$" specifies that the system does terminate but can change any variable of the state. So in fact, the terminating probabilistic specification substitution $\{p \mid v : \langle Q \rangle\}$ allows variables other than v to be changed (at most) $1 - p$ of the time.

Moreover, this behaviour also can be observed with our original probabilistic specification substitution, since the only difference with the terminating probabilistic specification substitution (beside the fact that the terminating version only applies for embedded predicate expectations) is the termination condition. This means we cannot guarantee that the specification change only the variables specified in the frame. In fact, a program such as

$$coinA := Head \quad {}_{\frac{1}{2}}\!\oplus \quad coinB := Tail$$

is a valid implementation of both programs

$$\frac{1}{2} \mid coinA : \langle coinA = Head \rangle$$

and

$$\left\{ \frac{1}{2} \mid coinA : \langle coinA = Head \rangle \right\} .$$

Recall our probabilistic specification substitution defined as "$A \mid v : B$" with the meaning that the substitution establishes the expectation of B (in the final set of distributions) with at least the expected value A (in the initial distribution) and "is constrained" to change variable in v only. Now we can have more precise understanding of our terminating probabilistic specification $\{p \mid v : \langle Q \rangle\}$: it specifies a terminating program which guarantees to establish the post condition Q and change only variables in v with probability of at least p, i.e. the other $1 - p$ of the time, other terminating behaviours are allowed, including changes made to variables not in v.

Changing variables outside the frame is in fact not something new. Consider the program (in the standard context) as follows:

$$\mathsf{false} \mid v : \mathsf{true} .$$

[8] This issue arose in our discussions with Prof. Steve Schneider at University of Surrey.

This program is in fact equivalent to abort and it is never guaranteed to terminate: even when it does, it can change other variables rather than variables in v.

We can define a special version of the terminating probabilistic specification substitution (we called this the restricted terminating probabilistic specification substitution) to restrict the changes made to the variables in the frame of the specification, as follows.

Definition 7 *Let p be a probabilistic expression over x and free from x_0; and let Q be a predicate defined over x_0, v and satisfying*

$$\forall\, x_0 \cdot (\exists\, v \cdot Q)\ .$$

The specification $\{\!|p \mid v : \langle Q\rangle|\!\}$ is defined by

$$\{\!|p \mid v : \langle Q\rangle|\!\} \quad \widehat{=} \quad 1 \mid v : \langle Q\rangle \quad {}_p\!\oplus \quad 1 \mid v : 1\ . \tag{5.27}$$

That is, the specification $\{\!|p \mid v : \langle Q\rangle|\!\}$ establishes Q with probability at least p; but even if it does not, it still terminates and changes only variables in v.[9]

Furthermore, we have a special version of the fundamental theorem corresponding to the above (restricted) specification as follows.

Theorem 7 *Let p be an expression over x and let Q be a predicate defined over x_0, v, and satisfying $\forall\, x_0 \cdot (\exists\, v \cdot Q)$; and let T be any program. For all such programs T, if*

$$(1 \mid v : 1) \quad \sqsubseteq \quad T\ ,\ \textit{i.e. } T \textit{ changes only variables in } v \textit{ and terminates},$$

and

$$(p \mid v : \langle Q\rangle) \quad \sqsubseteq \quad T$$

then

$$\{\!|p \mid v : \langle Q\rangle|\!\} \quad \sqsubseteq \quad T\ .$$

Proof. The proof of the theorem is similar to the proof in Sec. 5.7. Details can be found in Appendix C.

[9] The restriction in the frame is what distinguishes it from $\{p \mid v : \langle Q\rangle\}$. That restriction is made by changing the frame of the right-hand-side branch of the probabilistic choice in (5.27) from x to v.

The first issue with restricted terminating probabilistic specification substitutions is tool support. We have to check not only the refining program, i.e. T, terminates (which the original toolkit already generates proof obligations for), but also whether T makes changes only to variables within the frame. This can be easily done by syntactic checking through the program to discover which variables have been changed. We defer this as future work.

Moreover, the second issue is related to the completeness of probabilistic specification substitutions. In the standard context, every sequential program can be described by a single standard specification substitution (i.e. by specifying a pair of pre- and post-conditions). In the probabilistic context, not every program can be described by a probabilistic specification substitution. A simple example is a fair coin, i.e. a coin that returns $Head$ and $Tail$ with equal probability. This kind of system needs to be described by "multiple pre- and post-expectations". To establish each expectation, a program might have to modify different sets of variables (i.e. having a different frame for each post-expectation). We will explore this kind of system further in the next chapter.

5.9 Conclusions and Future Work

In this chapter, we proposed a second step in to the *pB* domain, by adding to the earlier proposed probabilistic machines (Sec. 4) the "superstructure" required for following the concept of the specify-refine-implement path embodied in the *B-Method*. We showed that the new construct is well defined and interacts properly with the other existing constructs in *B*, especially stepwise refinement. The case study of the Min-Cut algorithm turns out to be not completely trivial when it comes to formalisation [10].

Similar ideas have been presented by Neil White, but they are expressed in *Z*, in his MSc thesis at Oxford [67], even though he did not consider the extensions to terminating specifications. The reason for that is there are no compositions in *Z*.

Moreover, the *B* context provides a number of new challenges, some of which we have addressed here. The issue of *separation* of standard reasoning from probabilistic reasoning is (or will be) of crucial importance if *pB* is to handle devel-

[10] It was suggested to us by Annabelle McIver, based on a case-study of the same algorithm done in the theorem-proving environment Coq [29] by Prof. Christine Paulin-Mohring of the *LRI* in Paris.

opments of anything like the same size and scope as standard *B* does. The (unrestricted) "terminating" specification substitution (mentioned here in Sec. 5.7) will probably become the one used in practice.

The *B-Toolkit* has been modified to accommodate the new syntax for the unrestricted terminating specification substitution, and the example given in this chapter has been developed within the new *pB-Toolkit*.

The restricted terminating probabilistic specification substitution needs to have tool support, which will check the scope of the refining program. This can be done by a simple syntactic check for all operations to record the set of variables which have been changed by the substitution in the operation. We regard this as possible future work.

Chapter 6

Multiple Expectation Systems

6.1 Completeness of Probabilistic Specification Substitutions

In the last chapter, we introduced the notion of probabilistic specification substitutions and the corresponding fundamental theorem used to refine systems specified by such substitutions. We have also investigated and introduced a different variation of the probabilistic specification substitution to include the terminating condition. This inclusion allows probabilistic systems to be developed in layers. Introducing the probabilistic specification substitution is an important step which takes *pB* closer to the refinement framework which is so characteristic of the standard *B-Method*.

As we concluded in the last chapter, probabilistic specification substitutions are not in themselves complete: a system specified by probabilistic specification substitutions can only describe one "aspect" of its behaviours using a single pair of pre- and post-expectations. In general, however, a probabilistic system usually needs to be specified by more than one pair of pre- and post-expectations. Specifying a fair coin is an obvious example for this: we need to specify that the coin returns heads (at least) half of the time, and also that it returns tails (at least) half of the time.

Thus we need to have a generalised version of the substitution defined in the previous chapter in order to capture the behaviours of probabilistic systems in general. Moreover, we will need to develop the corresponding fundamental theorem

for the new substitution. Here, the framework that we concentrate on will be similar to the one in the previous chapter. The new substitution should fit snugly into the framework, i.e. it should allow systems to be developed in layers. We will use the example of "Duelling Cowboys" to illustrate our ideas.

This chapter is organised as follows: in Sec. 6.2, we define the syntax and give the semantics for "multiple probabilistic specification substitution"; in Sec. 6.3, we give the corresponding fundamental theorem and a simple example using the theorem; in Sec. 6.4, we give the case study of *Duelling Cowboys*, starting from a simple situation with two cowboys, and then extending that to the case where there are three; in Sec. 6.5, we propose possible changes to the *B-Toolkit* to accommodate the new substitution and its fundamental theorem; in Sec. 7, we conclude and discuss possible future work.

6.2 Multiple Probabilistic Specification Substitutions

In this section, we introduce the concept of multiple probabilistic specification substitutions and we give some examples to illustrate when and how we can use them.

6.2.1 Definition

We begin by considering the special case where the post-expectations are standard, i.e. they are all embedded predicates of the form $\langle Q \rangle$ where Q is a predicate of the state.

Recall the definition of a "multi-way probabilistic choice" which is the generalised version of a probabilistic choice [49]. It specifies a (probabilistic) choice over n alternatives, as follows:

$$(S_1 @ p_1 \mid S_2 @ p_2 \mid \cdots \mid S_n @ p_n)\ ;$$

if we write it vertically, we have

$$\left|\begin{array}{ll} S_1 & @ p_1 \\ S_2 & @ p_2 \\ \cdots & \\ S_n & @ p_n \end{array}\right. ,$$

in which in either form the sum of the probabilities p_i for $i \in (1..n)$ is no more than 1. The meaning of the substitution is that it will behave like S_i at least p_i of the time, for all of the i's simultaneously. The semantics of the substitution is defined with respect to an arbitrary post-expectation B as follows:

$$\left[\begin{array}{l|ll} S_1 & @p_1 \\ S_2 & @p_2 \\ \cdots & \\ S_n & @p_n\ , \end{array}\right] B \quad \equiv \quad \begin{array}{ll} & p_1 \times [S_1]\, B \\ + & p_2 \times [S_2]\, B \\ + & \cdots \\ + & p_n \times [S_n]\, B\ . \end{array}$$

More information on multi-way probabilistic choices can be found elsewhere [49].

We define (terminating) "multiple probabilistic specification substitutions", using multi-way probabilistic choices, in the following definition.

Definition 8 *For $i \in (1..n)$, let p_i be a probabilistic expression over the state x and free from x_0 and satisfying*

$$\sum_{i=1}^{n} p_i \leq 1\ ; \tag{6.1}$$

let Q_i be predicates defined over x_0, v (where v is a subset of x) and satisfying, for all Q_i, that we have

$$\forall\, x_0 \cdot (\exists\, v \cdot Q_i)\ . \tag{6.2}$$

Then the specification

$$v : \left|\begin{array}{l} \{p_1, \langle Q_1 \rangle\} \\ \{p_2, \langle Q_2 \rangle\} \\ \cdots \\ \{p_n, \langle Q_n \rangle\} \end{array}\right.$$

is defined by

$$v : \left|\begin{array}{l} \{p_1, \langle Q_1 \rangle\} \\ \{p_2, \langle Q_2 \rangle\} \\ \cdots \\ \{p_n, \langle Q_n \rangle\} \end{array}\right. \quad \widehat{=} \quad \left|\begin{array}{lll} (1 \mid v : \langle Q_1 \rangle) & @p_1 \\ (1 \mid v : \langle Q_2 \rangle) & @p_2 \\ \cdots & \\ (1 \mid v : \langle Q_n \rangle) & @p_n \\ (1 \mid x : 1) & @p_0\ , \end{array}\right. \tag{6.3}$$

where

$$p_0 = 1 - \sum_{i=1}^{n} p_i\ .$$

The definition states that with probability of at least p_i, the substitution behaves like program "$1 \;\mid\; v : \langle Q_i \rangle$" (for $i \in (1..n)$). Otherwise, the system still guarantees to terminate. Clearly, this is a generalised version of the terminating probabilistic specification substitution (if we have only one pair of pre- and post-expectation).

6.2.2 Examples

The first example that we consider is a fair coin. We can specify a fair coin using the substitution defined in the previous section as follows (assuming that the system only has one variable named $coin$):

$$coin : \left| \begin{array}{l} \{\frac{1}{2}, \langle coin = Head \rangle\} \\ \{\frac{1}{2}, \langle coin = Tail \rangle\} \; . \end{array} \right. \tag{6.4}$$

The specification specifies that it establishes $coin$ is $Head$ at least $\frac{1}{2}$ of the time and also establishes $coin$ is $Tail$ at least $\frac{1}{2}$ of the time.

We calculate the least guarantee probability that Prog. (6.4) establishes $coin$ is $Head$ according to the semantics as follows:

$$\left[coin : \left| \begin{array}{l} \{\frac{1}{2}, \langle coin = Head \rangle\} \\ \{\frac{1}{2}, \langle coin = Tail \rangle\} \end{array} \right. \right] \langle coin = Head \rangle$$

$\equiv$ multiple probabilistic specification substitution, see below

$$\left[\left| \begin{array}{ll} (1 \;\mid\; coin : \langle coin = Head \rangle) & @\; \frac{1}{2} \\ (1 \;\mid\; coin : \langle coin = Tail \rangle) & @\; \frac{1}{2} \end{array} \right. \right] \langle coin = Head \rangle$$

$\equiv$ multi-way probability choice

$$\begin{array}{rl} & \frac{1}{2} \;\times\; [1 \;\mid\; coin : \langle coin = Head \rangle] \;\; \langle coin = Head \rangle \\ + & \frac{1}{2} \;\times\; [1 \;\mid\; coin : \langle coin = Tail \rangle] \;\; \langle coin = Head \rangle \end{array}$$

$\equiv$ probabilistic specification substitutions

$$\begin{array}{l} \frac{1}{2} \;\times\; 1 \times [coin_0 := \; coin] \sqcap coin \cdot (\langle coin = Head \rangle \;\div\; \langle coin = Head \rangle) \\ +\frac{1}{2} \;\times\; 1 \times [coin_0 := \; coin] \sqcap coin \cdot (\langle coin = Tail \rangle \;\div\; \langle coin = Head \rangle) \end{array}$$

$\equiv$ the first minimum is 1 when $coin = Head$
the second minimum is 0 when $coin = Head$

$$\frac{1}{2} \times [coin_0 := \; coin]\, 1 \quad + \quad \frac{1}{2} \times [coin_0 := \; coin]\, 0$$

$\equiv$ $\frac{1}{2}$. simple substitutions and arithmetic

For the deferred judgement, zero probability is assigned to "$1 \mid coin : 1$", since the probabilities for $Head$ and $Tail$ sum to 1.

So the meaning of the substitution, as given by its semantics, shows that the probability of establishing $coin$ is $Head$ is indeed at least $\frac{1}{2}$. Similarly, the probability for Prog. (6.4) to establish $coin$ is $Tail$ is at least $\frac{1}{2}$.

We take another example where the state contains two coins, namely $coinA$ and $coinB$. We have the following specification:

$$(coinA, coinB) : \left| \begin{array}{l} \{\frac{1}{3}, \langle coinA = Head \rangle\} \\ \{\frac{1}{4}, \langle coinA = Tail \wedge coinB = Head \rangle\} \\ \{\frac{1}{4}, \langle coinA = Tail \wedge coinB = Tail \rangle\} \end{array} \right. , \tag{6.5}$$

which specifies that the substitution establishes $coinA$ is $Head$ at least $\frac{1}{3}$ of the time; $coinA$ is $Tail$ and $coinB$ is $Head$ at least $\frac{1}{4}$ of the time; and finally, $coinA$ and $coinB$ are both $Tail$ at least $\frac{1}{4}$ of the time.

Our intuition tells us that the guaranteed probability for Prog. (6.5) to establishes $coinA$ is $Tail$ is $\frac{1}{2}$ (the probability to establish $coinA$ is $Tail$ and $coinB$ is $Head$ plus the probability to establish $coinA$ is $Tail$ and $coinB$ is $Tail$). Again, we will use the semantics given in Def. 8 to calculate the probability as follows:

$$\left[(coinA, coinB) : \left| \begin{array}{l} \{\frac{1}{3}, \langle coinA = Head \rangle\} \\ \{\frac{1}{4}, \langle coinA = Tail \wedge coinB = Head \rangle\} \\ \{\frac{1}{4}, \langle coinA = Tail \wedge coinB = Tail \rangle\} \end{array} \right. \right] \langle coinA = Tail \rangle$$

$\equiv$ multiple probabilistic specification substitution, see below

$$\left[\begin{array}{ll} (1 \mid (coinA, coinB) : \langle coinA = Head \rangle) & @ \frac{1}{3} \\ (1 \mid (coinA, coinB) : \langle coinA = Tail \wedge coinB = Head \rangle) & @ \frac{1}{4} \\ (1 \mid (coinA, coinB) : \langle coinA = Tail \wedge coinB = Tail \rangle) & @ \frac{1}{4} \\ (1 \mid (coinA, coinB) : 1) & @ \frac{1}{6} \end{array} \right] \langle coinA = Tail \rangle$$

$\equiv$ multi-way probabilistic choice

$$
\begin{array}{rl}
 & \frac{1}{3}\times \quad [1 \mid (coinA, coinB) : \langle coinA = Head\rangle]\,\langle coinA = Tail\rangle \\
+ & \frac{1}{4}\times \quad [1 \mid (coinA, coinB) : \langle coinA = Tail \wedge coinB = Head\rangle] \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \langle coinA = Tail\rangle \\
+ & \frac{1}{4}\times \quad [1 \mid (coinA, coinB) : \langle coinA = Tail \wedge coinB = Tail\rangle] \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \langle coinA = Tail\rangle \\
+ & \frac{1}{6}\times \quad [1 \mid (coinA, coinB) : 1]\,\langle coinA = Tail\rangle
\end{array}
$$

$\equiv$ probabilistic specification substitutions

$$
\begin{array}{rl}
 & \frac{1}{3}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB] \\
 & \qquad\qquad\qquad\qquad \sqcap(coinA, coinB) \cdot (\langle coinA = Tail\rangle \div \langle coinA = Head\rangle) \\
+ & \frac{1}{4}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB] \\
 & \sqcap(coinA, coinB) \cdot (\langle coinA = Tail\rangle \div \langle coinA = Tail \wedge coinB = Head\rangle) \\
+ & \frac{1}{4}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB] \\
 & \sqcap(coinA, coinB) \cdot (\langle coinA = Tail\rangle \div \langle coinA = Tail \wedge coinB = Tail\rangle) \\
+ & \frac{1}{6}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB] \\
 & \qquad\qquad\qquad\qquad \sqcap(coinA, coinB) \cdot (\langle coinA = Tail\rangle \div 1)
\end{array}
$$

$\equiv$ the first minimum is 0 when $coinA = Head$
the second minimum is 1 when $coinA = Tail \wedge coinB = Head$
the third minimum is 1 when $coinA = Tail \wedge coinB = Tail$
the fourth minimum is 0 when $coinA = Head$

$$
\begin{array}{rl}
 & \frac{1}{3}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB]\,0 \\
+ & \frac{1}{4}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB]\,1 \\
+ & \frac{1}{4}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB]\,1 \\
+ & \frac{1}{6}\times \quad 1 \times [coinA_0, coinB_0 := coinA, coinB]\,0
\end{array}
$$

$\equiv$ $\frac{1}{2}$, simple substitutions and arithmetic

which is consistent with our intuition about the program.

For the deferred judgement, the probability assigned to "$1 \mid (coinA, coinB) : 1$" is

$$1 - \left(\frac{1}{3} + \frac{1}{4} + \frac{1}{4}\right) = \frac{1}{6}\,.$$

6.3 The Multiple Probabilistic Fundamental Theorem

In this section, we formulate the fundamental theorem corresponding to the substitution introduced in the previous section, and look at an example showing how to use the theorem. We are considering a special kind of multiple probabilistic specification substitution where the post-conditions are disjoint.

6.3.1 The Fundamental Theorem

The theorem is an extension of the fundamental theorem for terminating probabilistic specification substitutions, and can be stated as follows:

Theorem 8 *For $i \in (1..n)$, let p_i be probabilistic expressions over x and free from x_0 and satisfying*

$$\sum_{i=1}^{n} p_i \leq 1 \,. \tag{6.6}$$

Let Q_i be predicates defined over x_0, v and satisfying, for all predicates Q_i, that we have

$$\forall\, x_0 \cdot (\exists\, v \cdot Q_i) \,. \tag{6.7}$$

Moreover, all those predicates are pairwise disjoint [1]*, i.e. for any pair Q_i and Q_j, where $i \neq j$, we have*

$$Q_i \wedge Q_j = \mathit{false} \,. \tag{6.8}$$

For all programs T, if

- *T terminates, i.e.*

$$(1 \mid x : 1) \quad \sqsubseteq \quad T \,, \tag{6.9}$$

  and

- *for all $i \in (1..n)$,*

$$(p_i \mid v : \langle Q_i \rangle) \quad \sqsubseteq \quad T \tag{6.10}$$

then we have

$$v : \left| \begin{array}{l} \{p_1, \langle Q_1 \rangle\} \\ \{p_2, \langle Q_2 \rangle\} \\ \cdots \\ \{p_n, \langle Q_n \rangle\} \end{array} \right. \quad \sqsubseteq \quad T \,. \tag{6.11}$$

[1] This disjointness condition is the extra condition for the soundness of the fundamental theorem.

Proof. Let E be an arbitrary expectation over x. For any x_0, we have from arithmetic that

$$E \quad \Leftarrow \quad \begin{array}{ll} \sqcap x \cdot (E) \times 1 & + \\ (\sqcap x \cdot (E \div \langle Q_1^w \rangle) - \sqcap x \cdot (E)) \times \langle Q_1^w \rangle & + \\ (\sqcap x \cdot (E \div \langle Q_2^w \rangle) - \sqcap x \cdot (E)) \times \langle Q_2^w \rangle & + \\ \cdots & + \\ (\sqcap x \cdot (E \div \langle Q_n^w \rangle) - \sqcap x \cdot (E)) \times \langle Q_n^w \rangle & \end{array} \tag{6.12}$$

where $Q_i^w \;\widehat{=}\; Q_i \wedge w = w_0$, for $i \in (1..n)$. We have the above inequality because the embedded predicates $\langle Q_i^w \rangle$ and $\langle w = w_0 \rangle$ can have values only 1 or 0. In each case, it can be proved that the inequality (6.12) holds, as shown below.

- If $w \neq w_0$, the right-hand side of (6.12) is $\sqcap x \cdot (E)$, which is everywhere no more than E.
- If Q_i^w holds for a particular i, with the condition (6.8) Q_j^w is false for any $j \neq i$. Given the non-emptiness condition (6.7), the right-hand side of (6.12) is $\sqcap x \cdot (E \div \langle Q_i^w \rangle)$ which is everywhere no more than E, given that Q_i^w holds.
- If $w = w_0 \wedge \neg(Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n)$, the right-hand side of (6.12) is $\sqcap x \cdot (E)$, which is everywhere no more than E.

First, we can estimate the pre-expectation of T with respect to E, since for all x_0 we have:

$$\begin{array}{ll} & [T]\, E \\ \Leftarrow & \qquad\qquad\qquad\qquad\qquad\qquad \text{(6.12) and monotonicity of } T \\ & [T] \left(\begin{array}{ll} \sqcap x \cdot (E) \times 1 & + \\ (\sqcap x \cdot (E \div \langle Q_1^w \rangle) - \sqcap x \cdot (E)) \times \langle Q_1^w \rangle & + \\ (\sqcap x \cdot (E \div \langle Q_2^w \rangle) - \sqcap x \cdot (E)) \times \langle Q_2^w \rangle & + \\ \cdots & + \\ (\sqcap x \cdot (E \div \langle Q_n^w \rangle) - \sqcap x \cdot (E)) \times \langle Q_n^w \rangle & \end{array} \right) \\ \Leftarrow & \qquad\qquad\qquad\qquad\qquad\qquad \text{general sublinearity of } T \text{ (see (5.24) in Sec. 5.7)} \\ & \begin{array}{ll} \sqcap x \cdot (E) \times [T]1 & + \\ (\sqcap x \cdot (E \div \langle Q_1^w \rangle) - \sqcap x \cdot (E)) \times [T]\langle Q_1^w \rangle & + \\ (\sqcap x \cdot (E \div \langle Q_2^w \rangle) - \sqcap x \cdot (E)) \times [T]\langle Q_2^w \rangle & + \\ \cdots & + \\ (\sqcap x \cdot (E \div \langle Q_n^w \rangle) - \sqcap x \cdot (E)) \times [T]\langle Q_n^w \rangle\,. & \end{array} \end{array}$$

Hence (since x_0 is unconstrained), in particular,

$$
\begin{array}{ll}
& [T]E \\
\Leftarrow & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_0 \text{ is given a value } x \\
& [x_0 := x] \begin{pmatrix} \sqcap x \cdot (E) \times [T]1 & + \\ (\sqcap x \cdot (E \div \langle Q_1^w \rangle) - \sqcap x \cdot (E)) \times [T]\langle Q_1^w \rangle & + \\ (\sqcap x \cdot (E \div \langle Q_2^w \rangle) - \sqcap x \cdot (E)) \times [T]\langle Q_2^w \rangle & + \\ \cdots & + \\ (\sqcap x \cdot (E \div \langle Q_n^w \rangle) - \sqcap x \cdot (E)) \times [T]\langle Q_n^w \rangle & \end{pmatrix} \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{simple substitution } x_0 := x \\
& \begin{array}{ll}
[x_0 := x](\sqcap x \cdot (E) \times [T]1) & + \\
[x_0 := x]\,(\sqcap x \cdot (E \div \langle Q_1^w \rangle) - \sqcap x \cdot (E)) & \\
\qquad \times [x_0 := x][T]\langle Q_1^w \rangle & + \\
[x_0 := x]\,(\sqcap x \cdot (E \div \langle Q_2^w \rangle) - \sqcap x \cdot (E)) & \\
\qquad \times [x_0 := x][T]\langle Q_2^w \rangle & + \\
\cdots & + \\
[x_0 := x]\,(\sqcap x \cdot (E \div \langle Q_n^w \rangle) - \sqcap x \cdot (E)) & \\
\qquad \times [x_0 := x][T]\langle Q_n^w \rangle &
\end{array} \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_0 \text{ is a fresh variable and simple substitution} \\
& \begin{array}{ll}
[x_0 := x] \sqcap x \cdot (E) \;\times\; [x_0 := x][T]1 & + \\
[x_0 := x]\,(\sqcap x \cdot (E \div \langle Q_1^w \rangle) - \sqcap x \cdot (E)) & \\
\qquad \times [x_0 := x][T]\langle Q_1^w \rangle & + \\
[x_0 := x]\,(\sqcap x \cdot (E \div \langle Q_2^w \rangle) - \sqcap x \cdot (E)) & \\
\qquad \times [x_0 := x][T]\langle Q_2^w \rangle & + \\
\cdots & + \\
[x_0 := x]\,(\sqcap x \cdot (E \div \langle Q_n^w \rangle) - \sqcap x \cdot (E)) & \\
\qquad \times [x_0 := x][T]\langle Q_n^w \rangle &
\end{array} \\
\Leftarrow & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{refinement assumption} \\
& \begin{array}{ll}
[x_0 := x] \sqcap x \cdot (E) \times 1 & + \\
[x_0 := x]\,(\sqcap x \cdot (E \;\div\; \langle Q_1^w \rangle) - \sqcap x \cdot (E)) \times p_1 & + \\
[x_0 := x]\,(\sqcap x \cdot (E \;\div\; \langle Q_2^w \rangle) - \sqcap x \cdot (E)) \times p_2 & + \\
\cdots & + \\
[x_0 := x]\,(\sqcap x \cdot (E \;\div\; \langle Q_n^w \rangle) - \sqcap x \cdot (E)) \times p_n &
\end{array}
\end{array}
$$

$\equiv$ simple substitution and arithmetic [2]

$$\begin{array}{ll} [x_0 := x] \sqcap x \cdot (E) \times p_0 & + \\ [x_0 := x] \sqcap x \cdot (E \div \langle Q_1^w \rangle) \times p_1 & + \\ [x_0 := x] \sqcap x \cdot (E \div \langle Q_2^w \rangle) \times p_2 & + \\ \cdots & + \\ [x_0 := x] \sqcap x \cdot (E \div \langle Q_n^w \rangle) \times p_n & \end{array}$$

$\equiv$ definition of probabilistic specification substitution

$$\begin{array}{ll} [1 \mid x : 1]E \times p_0 & + \\ [1 \mid v : \langle Q_1 \rangle]E \times p_1 & + \\ [1 \mid v : \langle Q_2 \rangle]E \times p_2 & + \\ \cdots & + \\ [1 \mid v : \langle Q_n \rangle]E \times p_n & \end{array}$$

$\equiv$ multi-way probabilistic choice substitution

$$\left[\begin{array}{l|ll} & (1 \mid x : 1) & @p_0 \\ & (1 \mid v : \langle Q_1 \rangle) & @p_1 \\ & (1 \mid v : \langle Q_2 \rangle) & @p_2 \\ & \cdots & \\ & (1 \mid v : \langle Q_n \rangle) & @p_n \end{array}\right] E$$

$\equiv$ definition of multiple terminating probabilistic specification substitution

$$\left[v : \left| \begin{array}{l} \{p_1, \langle Q_1 \rangle\} \\ \{p_2, \langle Q_2 \rangle\} \\ \cdots \\ \{p_n, \langle Q_n \rangle\} \end{array} \right. \right] E$$

Because E is arbitrary, therefore

$$v : \left| \begin{array}{l} \{p_1, \langle Q_1 \rangle\} \\ \{p_2, \langle Q_2 \rangle\} \\ \cdots \\ \{p_n, \langle Q_n \rangle\} \end{array} \right. \quad \sqsubseteq \quad T \; .$$

[2] We have defined

$$p_0 = 1 - \sum_{i=1}^{n} p_i \; .$$

6.3.2 Examples

We consider a particular program which is the refinement for Prog. (6.5).

Let

$$T \mathrel{\widehat{=}} \quad coinA := Head \;\; {}_{\frac{1}{2}}\!\oplus \left(\begin{array}{l} coinA, coinB := Tail, Head \\ {}_{\frac{1}{2}}\!\oplus \\ coinA, coinB := Tail, Tail \end{array} \right)$$

According to the fundamental theorem in previous section, we need to prove that

$$1 \mid (coinA, coinB) : 1 \quad \sqsubseteq T \tag{6.13}$$

$$\frac{1}{3} \mid (coinA, coinB) : \langle coinA = Head \rangle \quad \sqsubseteq T \tag{6.14}$$

$$\frac{1}{4} \mid (coinA, coinB) : \langle coinA = Tail \wedge coinB = Head \rangle \quad \sqsubseteq T \tag{6.15}$$

$$\frac{1}{4} \mid (coinA, coinB) : \langle coinA = Tail \wedge coinB = Tail \rangle \quad \sqsubseteq T \tag{6.16}$$

Proof. For (6.13), clearly, T terminates and makes changes to variables $coinA$ or $coinB$ only.

We prove (6.14) by using the probabilistic fundamental theorem in Sec. 5.3.2. Consider the post-expectation $\langle coinA = Head \rangle$.

$$[coinA_0, coinB_0 := coinA, coinB]\,[T]\,\langle coinA = Head \rangle$$

$\equiv$ definition of T

$$[coinA_0, coinB_0 := coinA, coinB] \left[\begin{array}{l} coinA := Head \\ {}_{\frac{1}{2}}\!\oplus \\ \left(\begin{array}{l} coinA, coinB := Tail, Head \\ {}_{\frac{1}{2}}\!\oplus \\ coinA, coinB := Tail, Tail \end{array} \right) \end{array} \right] \langle coinA = Head \rangle$$

$\equiv$ probabilistic choice substitution
there are no $coinA_0$ and $coinB_0$ in T

$$\begin{aligned} & \tfrac{1}{2} \times \quad [coinA := Head] \langle coinA = Head \rangle \\ + \quad & \left(1 - \tfrac{1}{2}\right) \times \left[\begin{array}{l} coinA, coinB := Tail, Head \\ {}_{\frac{1}{2}}\!\oplus \\ coinA, coinB := Tail, Tail \end{array} \right] \langle coinA = Head \rangle \end{aligned}$$

$$
\begin{array}{ll}
\equiv & \text{simple substitution and arithmetic} \\
& \quad \tfrac{1}{2} \times \quad \langle Head = Head \rangle \\
& + \quad \tfrac{1}{2} \times \left[\begin{array}{l} coin A, coin B := \ Tail, Head \\ \tfrac{1}{2} \oplus \\ coin A, coin B := \ Tail, Tail \end{array} \right] \langle coin A = Head \rangle \\
\\
\equiv & \text{probabilistic choice substitution and embedded predicate} \\
& \quad \tfrac{1}{2} \times \quad 1 \\
& + \quad \tfrac{1}{2} \times \left(\begin{array}{c} \tfrac{1}{2} \times \quad [coin A, coin B := \ Tail, Head] \langle coin A = Head \rangle \\ + \\ \left(1 - \tfrac{1}{2}\right) \times [coin A, coin B := \ Tail, Tail] \ \langle coin A = Head \rangle \end{array} \right) \\
\\
\equiv & \text{simple substitution and arithmetic} \\
& \quad \tfrac{1}{2} \quad + \quad \tfrac{1}{2} \times \left(\tfrac{1}{2} \times \langle Tail = Head \rangle \ + \ \tfrac{1}{2} \times \langle Tail = Head \rangle \right) \\
\\
\equiv & \quad \tfrac{1}{2} \quad + \quad \tfrac{1}{2} \times \left(\tfrac{1}{2} \times 0 \ + \ \tfrac{1}{2} \times 0 \right) \qquad \text{embedded predicate} \\
\equiv & \quad \tfrac{1}{2} \, . \qquad \text{arithmetic}
\end{array}
$$

We already proved that

$$\frac{1}{3} \quad \Rrightarrow \quad [coin A_0, coin B_0 := \ coin A, coin B] \, [T] \, \langle coin A = Head \rangle \ ,$$

hence by the probabilistic fundamental theorem, (6.14) holds.

Proofs of (6.15) and (6.16) are similarly trivial and will not be mentioned here.

Hence in order to prove the refinement of multiple probabilistic specification substitutions, we can separate and prove the refinement for each pair of pre- and post-expectation independently (using the probabilistic fundamental theorem described in Sec. 5.3.2).

6.4 Case Study: The Duelling Cowboys

In this section, we show how multiple probabilistic specification substitutions can be applied in order to specify systems with probabilistic properties. Moreover, we show that we can prove the refinement of such systems to executable code using the corresponding fundamental theorem.

6.4.1 Example of Two Duelling Cowboys

The description of the problem is as follows. There are two cowboys X and Y fighting a duel. They take turns to shoot at each other. In each shot, the probability for X to hit his opponent is $\frac{2}{3}$, whereas the probability for Y is $\frac{1}{2}$. Assuming that X has the advantage of shooting first, what are the (least) survival probabilities for both cowboys? We start with some informal reasonings of the problem, and follow by formalising the system in *pB*.

Informal Reasoning

We reason about the probability of X first. Let the probability that X survives the shooting when he has the chance to go first be $p_{XY}(X)$. Here, the subscript XY indicates that there are two cowboys, X and Y, and X is going to shoot first. We have the equation

$$p_{XY}(X) \quad = \quad \frac{2}{3} \times 1 \; + \; \frac{1}{3} \times p_{YX}(X)\,, \tag{6.17}$$

where $p_{YX}(X)$ is the chance that X survives the game if Y gets the chance to shoot X first. This is because with probability $\frac{2}{3}$, X is successful and he survives the game (with probability 1). With probability $\frac{1}{3}$, he misses the shot and Y now has the turn to shoot. With similar reasoning, we have:

$$p_{YX}(X) \quad = \quad \frac{1}{2} \times 0 \; + \; \frac{1}{2} \times p_{XY}(X)\,. \tag{6.18}$$

Solving the set of equations (6.17) and (6.18) for $p_{XY}(X)$ and $p_{YX}(X)$, we have $p_{XY}(X)$ is $\frac{4}{5}$. So the probability that cowboy X survives the duel is at least $\frac{4}{5}$ (since there is no non-determinism in this game, this will be in fact the exact probability).

With similar reasoning, the least probability that cowboy Y survives the shooting is $\frac{1}{5}$. This is in fact the complement of probability of X since the duelling is deterministic.

Formal Development of Two Duelling Cowboys

We use multiple probabilistic specification substitutions to specify the two duelling cowboy system. Let s be the cowboy surviving the duel, so that the specification is as shown in Fig. 6.4.1 on the following page. (We use the term "post-hoc" here

$$s \longleftarrow \textbf{TwoCowboyXY} \quad \widehat{=} \quad s : \left| \begin{array}{l} \{\frac{4}{5}, \langle s = X \rangle\} \\ \{\frac{1}{5}, \langle s = Y \rangle\} \end{array} \right.$$

Figure 6.1: "Post-hoc" specification of Two Duelling Cowboys

because for the sake of our example we have in fact derived the specification from the implementation of the problem, albeit informally.)

The operation (namely TwoCowboyXY) has no inputs and has one output representing the cowboy surviving the duel. Here, we use the *pGSL* form for the operation [3]. The operation specifies that (according to the meaning of multiple probabilistic specification substitutions) the probability that cowboy X survives the game is (at least) $\frac{4}{5}$, and that the probability for Y is (at least) $\frac{1}{5}$.

The implementation of the two duelling cowboys can be seen in Fig. 6.2. The implementation uses two local variables: t to record the cowboy whose turn it is to shoot, and n for the number of cowboys still alive (so that n is either 1 or 2). The game starts with cowboy X having the turn to shoot first, and both cowboys are alive initially.

Then they start shooting at each other in turn. This is implemented by a *WHILE* loop, with the guard indicating that both cowboys are alive. The body of the loop is a conditional substitution. The first branch of the *IF* clause models the case where cowboy X shoots: with probability $\frac{2}{3}$ X is successful and he will be the only surviving cowboy (and n decreases accordingly); otherwise, Y will have the turn to shoot. The *ELSE* branch is complementary in the obvious way for Y. We leave the details about expectations of the loops for later reasoning.

We reason about the termination of the loop first: in fact the loop is almost-certain to terminate (in the sense described in Sec. 3.2.1). The chosen variant is the number of surviving cowboys, n, which is either 1 or 2 (i.e. bounded above by 2). Moreover, because both probabilities of success for two cowboys are non-zero, it can be easily seen that there is a non-zero chance that the variant decreases from 2 to 1 (if one of the cowboys is successful). So by the almost-certain variant rule in Sec. 3.2.4, the loop terminates with probability one.

In order to prove the refinement, we apply the multiple probabilistic fundamen-

[3] Later, we will discuss the *pAMN* form for the substitution.

```
s ⟵ TwoCowboyXY  ≙
    VAR t, n IN
        t := X ; s := X ; n := 2 ;
        WHILE n = 2 DO
            IF t = X THEN
                PCHOICE 2//3 OF s := X ; n := 1
                OR t := Y
                END
            ELSE
                PCHOICE 1//2 OF s := Y ; n := 1
                OR t := X
                END
            END
        BOUND 2
        VARIANT n
        INVARIANT
            n ∈ 1..2
        EXPECTATIONS ···
        END
    END
```

Figure 6.2: Implementation of Two Duelling Cowboys

tal theorem. We need to prove that the implementation in fact refines

$$\frac{4}{5} \mid s : \langle s = X \rangle \tag{6.19}$$

and

$$\frac{1}{5} \mid s : \langle s = Y \rangle \; ; \tag{6.20}$$

we leave aside the fact that it terminates and makes changes to variable s only (which is trivial and we will not reason about).

Here, we prove the refinement for (6.19) using the following expectation (probabilistic invariant) for the loop:

$$E_1 \quad \equiv \quad \langle s = X \wedge n = 1 \rangle + \langle n = 2 \wedge t = X \rangle \times \frac{4}{5} + \langle n = 2 \wedge t = Y \rangle \times \frac{2}{5} \; .$$

According to Sec. 5.4.2, we have to prove that

$$\frac{4}{5} \Rrightarrow [s_0 := s] \, [TwoCowboyXY] \, \langle s = X \rangle \; ,$$

which is equivalent to

$$\frac{4}{5} \Rrightarrow [TwoCowboyXY] \, \langle s = X \rangle \; ,$$

since there are no s_0 variables in the operation TwoCowboyXY. (Notice that we mean the implementation version of the operation.)

With respect to Sec. 5.4.2 concerning the partial correctness of probabilistic loops, we have the following information for the *WHILE* loop in the implementation of operation TwoCowboyXY.

- The standard invariant is $I_1 \quad \widehat{=} \quad n \in 1..2$.
- The guard for the loop is $G_1 \quad \widehat{=} \quad n = 2$.
- The body of the loop is

$$S_1 \quad \widehat{=} \quad \mathit{IF} \cdots \mathit{THEN} \cdots \mathit{ELSE} \cdots \mathit{END} \; .$$

- The expectation (probabilistic invariant) is

$$E_1 \quad \widehat{=} \quad \langle s = X \wedge n = 1 \rangle + \langle n = 2 \wedge t = X \rangle \times \frac{4}{5} + \langle n = 2 \wedge t = Y \rangle \times \frac{2}{5} \; .$$

- The post-condition Q_1 is the constant predicate true.
- The post-expectation is $B_1 \;\widehat{=}\; \langle s = X \rangle$.

Here, we concentrate on proving the obligations related to the expectations only. Proving the maintenance of the standard invariant for the loop is trivial.

1. On termination, the expectation E_1 is at least the post-expectation B_1, i.e. if $\neg G_1 \wedge I_1$ then $E_1 \Rrightarrow B_1$. We have

$$
\begin{array}{lll}
 & E_1 & \\
\equiv & & \text{definition of } E_1 \\
 & \langle s = X \wedge n = 1 \rangle + \langle n = 2 \wedge t = X \rangle \times \tfrac{4}{5} + \langle n = 2 \wedge t = Y \rangle \times \tfrac{2}{5} & \\
\equiv & & \neg G_1 \wedge I_1 \;\Rightarrow\; n = 1 \\
 & \langle s = X \rangle + \langle \mathsf{false} \rangle \times \tfrac{4}{5} + \langle \mathsf{false} \rangle \times \tfrac{2}{5} & \\
\equiv & \langle s = X \rangle & \text{embedded predicate and arithmetic} \\
\equiv & B_1 \,, & \text{definition of } B_1
\end{array}
$$

   which is what we needed to prove.

2. The expectation E_1 is "maintained" (does not decrease) during the execution of the loop, i.e. if $G_1 \wedge I_1$ then $E_1 \Rrightarrow [S_1]\, E_1$. We start the calculation from the right-hand side:

$$
\begin{array}{lll}
 & [S_1]\, E_1 & \\
\equiv & [t = X \implies S_{1a} \;[\!]\; t \neq X \implies S_{1b}]\, E_1 & \text{definition of } S_1 \text{ [4]} \\
\equiv & & \text{non-deterministic choice substitution} \\
 & \min \begin{array}{l} [t = X \Longrightarrow S_{1a}]\, E_1 \\ [t \neq X \Longrightarrow S_{1b}]\, E_1 \end{array} &
\end{array}
$$

[4] S_{1a} and S_{1b} are the corresponding probabilistic choice branches in the body of the *WHILE* loop. Also note that we use the *pGSL* form of substitution here.

$$\equiv \quad \min \begin{array}{l} \frac{1}{\langle t=X\rangle} \times [S_{1a}]\, E_1 \\ \frac{1}{\langle t\neq X\rangle} \times [S_{1b}]\, E_1 \ . \end{array} \qquad \text{guarded substitutions}$$

$$\equiv \qquad \text{details of the calculations are shown in Appendix D.1}$$

$$\tfrac{1}{\langle t=X\rangle} \times \tfrac{4}{5} \quad \min \quad \tfrac{1}{\langle t\neq X\rangle} \times \tfrac{2}{5}$$

$$\equiv \quad \langle t = X\rangle \times \tfrac{4}{5} \quad + \quad \langle t \neq X\rangle \times \tfrac{2}{5} \qquad \text{arithmetic, see below}$$

For the deferred judgement, when $t = X$, the value of $\frac{1}{\langle t\neq X\rangle}$ is infinity and the minimum will ignore that. The same occurs when $t \neq X$.

When $G_1 \wedge I_1$ holds, we have $n = 2$, hence

$$E_1$$

$$\equiv \qquad \text{definition of } E_1$$

$$\langle s = X \wedge n = 1\rangle + \langle n = 2 \wedge t = X\rangle \times \tfrac{4}{5} + \langle n = 2 \wedge t = Y\rangle \times \tfrac{2}{5}$$

$$\equiv \qquad \text{we have } n = 2$$

$$\langle s = X \wedge 2 = 1\rangle + \langle 2 = 2 \wedge t = X\rangle \times \tfrac{4}{5} + \langle 2 = 2 \wedge t = Y\rangle \times \tfrac{2}{5}$$

$$\equiv \qquad \text{logic, embedded predicates and arithmetic}$$

$$\langle t = X\rangle \times \tfrac{4}{5} \quad + \quad \langle t \neq X\rangle \times \tfrac{2}{5}$$

$$\equiv \quad [S_1]\, E_1 \ . \qquad \text{calculation above}$$

Thus we have proved that the expectation E_1 does not decrease during the execution of the loop.

3. The expectation E_1 is "established" at the beginning of the loop, i.e.

$$\frac{4}{5} \Rrightarrow [t := X; s := X; n := 2]\, E_1 \ .$$

We begin the proof from the right-hand side:

$$[t :=\ X; s :=\ X; n :=\ 2]\, E_1$$

$\equiv$ definition of E_1

$$[t :=\ X; s :=\ X; n :=\ 2]$$
$$\langle s = X \wedge n = 1\rangle + \langle n = 2 \wedge t = X\rangle \times \tfrac{4}{5} + \langle n = 2 \wedge t =\ Y\rangle \times \tfrac{2}{5}$$

$\equiv$ sequential and simple substitution

$$\langle X = X \wedge 2 = 1\rangle + \langle 2 = 2 \wedge X = X\rangle \times \tfrac{4}{5} + \langle 2 = 2 \wedge X =\ Y\rangle \times \tfrac{2}{5}$$

$\equiv$ $\frac{4}{5}$, embedded predicate and arithmetic

which is what we needed to prove.

Hence we have proved that

$$(\frac{4}{5} \mid s : \langle s = X\rangle) \sqsubseteq \mathsf{TwoCowboyXY} .$$

Similarly (but with a different expectation —probabilistic invariant— for loop), we can prove that

$$(\frac{1}{5} \mid s : \langle s =\ Y\rangle) \sqsubseteq \mathsf{TwoCowboyXY} ,$$

as required.

6.4.2 Example of Three Duelling Cowboys

In the previous example, we showed how the multiple probabilistic specification substitution can be used to specify systems with probabilistic properties. We also showed that we can implement the specification and prove the correctness of the refinement process to code.

We extend the example further to see how the multiple probabilistic specification substitutions fit into the framework of *B*, i.e. development via layers. The example is extended to cover the case where there are three cowboys instead of just two.

Assuming that beside X and Y, we have another cowboy Z, whose accuracy rate is $\frac{1}{3}$. They shoot at each other with the turns now following the sequence $X, Y, Z, X, Y, Z, \ldots$. Also, for simplicity, we assume that if all of them are alive

then any cowboy will choose to shoot the person who gets the turn to shoot next, e.g. X will shoot Y first, Y will shoot Z first, etc. Furthermore, if one cowboy has died, his turn will be passed to the next one in the sequence, e.g. if X is successful when shooting at Y, after that, it is Z who shoots. The same question is asked now, that is what are the survival chances for the cowboys, assuming that X has the turn to fire first.

Informal Reasoning

We reason about the probability of X first; the probabilities of Y and Z can be calculated in a similar way. Let the (least) probability that X survives the shooting when he has the chance to go first be $p_{XYZ}(X)$. Here the subscript XYZ indicates the sequence of shooting, as before. We have the equation

$$p_{XYZ}(X) \quad = \quad \frac{2}{3} \times p_{ZX}(X) \; + \; \frac{1}{3} \times p_{YZX}(X) \, , \tag{6.21}$$

where $p_{ZX}(X)$ is the (least) probability that X survives when there are only X and Z and Z has the chance to shoot first. Similarly, $p_{YZX}(X)$ is the probability that X survives when all of them are alive and Y has the turn to fire. We calculate the probability $p_{ZX}(X)$ using the technique of the previous section and find it to be $\frac{4}{7}$. Hence (6.21) can be rewritten as

$$p_{XYZ}(X) \quad = \quad \frac{8}{21} \; + \; \frac{1}{3} \times p_{YZX}(X) \, . \tag{6.22}$$

Similarly, we have the equations for $p_{YZX}(X)$ and $p_{ZXY}(X)$ as follows:

$$p_{YZX}(X) \quad = \quad \frac{2}{5} \; + \; \frac{1}{2} \times p_{ZXY}(X) \tag{6.23}$$

$$p_{ZXY}(X) \quad = \quad \frac{2}{3} \times p_{XYZ}(X) \, . \tag{6.24}$$

Solving the set of equations (6.22), (6.23) and (6.24), we have results that

$$p_{XYZ}(X) = \frac{81}{140} \, , \; p_{YZX}(X) = \frac{83}{140} \, , \; p_{ZXY}(X) = \frac{27}{70} \, .$$

So the probability that X survives the duelling with three cowboys is $\frac{81}{140}$. (Since the game is still deterministic, this is again the exact probability.)

With similar reasoning for Y and Z, the probability that Y survives the game is $\frac{27}{320}$ and the probability that Z survives is $\frac{151}{448}$.

$$s \longleftarrow \textbf{ThreeCowboyXYZ} \quad \widehat{=} \quad s : \left| \begin{array}{l} \{\frac{81}{140}, \langle s = X \rangle\} \\ \{\frac{27}{320}, \langle s = Y \rangle\} \\ \{\frac{151}{448}, \langle s = Z \rangle\} \end{array} \right.$$

Figure 6.3: "Post-hoc" specification of Three Duelling Cowboys

Formal Development of Three Duelling Cowboys

The specification of three duelling cowboys example can be seen in Fig. 6.3. The specification uses the notion of multiple probabilistic specification substitutions to define the probability for each cowboy to survive the game according to the values calculated in the previous section.

The implementation of the system can be seen in Fig. 6.4 on the following page. Again, we use two local variables: t to record the cowboy whose turn it is to shoot, and n for the number of cowboys still alive. Furthermore, we use specifications that model the behaviour when the duelling happens between two cowboys only (i.e. when one of them already been killed). Operation TwoCowboyXY is described in the previous section. Operation TwoCowboyZX is when the duelling happens between X and Z but Z has the chance to fire first, and similarly we have operation TwoCowboyYZ. The specifications of operations TwoCowboyYZ and TwoCowboyZX are as follows:

$$s \longleftarrow \mathsf{TwoCowboyYZ} \quad \widehat{=} \quad s : \left| \begin{array}{l} \{\frac{3}{4}, \langle s = Y \rangle\} \\ \{\frac{1}{4}, \langle s = Z \rangle\} \end{array} \right. \tag{6.25}$$

and

$$s \longleftarrow \mathsf{TwoCowboyZX} \quad \widehat{=} \quad s : \left| \begin{array}{l} \{\frac{3}{7}, \langle s = Z \rangle\} \\ \{\frac{4}{7}, \langle s = X \rangle\} \end{array} \right. \tag{6.26}$$

In the implementation, X has the advantage of shooting first, and n starts from 3. The duelling is modelled as an *WHILE* loop, which continues to iterate until there is only one cowboy surviving. A conditional substitution is used to implement different cases when each cowboy has the turn to shoot.

The first branch is for the case when X fires at Y (according to the earlier assumption): with probability $\frac{2}{3}$, he is successful and the duelling happens between him and Z, but Z has the turn to fire first; otherwise (with probability $\frac{1}{3}$), X misses the shot and Y now has the turn to shoot. Other branches which implement the cases where Y or Z has the turn to fire are similar.

```
s ⟵ ThreeCowboyXYZ ≙
    VAR t, n IN
        t := X ; s := X ; n := 3 ;
        WHILE n = 3 DO
            IF t = X THEN
                PCHOICE 2//3 OF s ⟵ TwoCowboyZX ; n := 1
                OR t := Y
                END
            ELSIF t = Y THEN
                PCHOICE 1//2 OF s ⟵ TwoCowboyXY ; n := 1
                OR t := Z
                END
            ELSE
                PCHOICE 1//3 OF s ⟵ TwoCowboyYZ ; n := 1
                OR t := X
                END
            END
            BOUND 3
            VARIANT n
            INVARIANT
                n ∈ {1, 3}
            EXPECTATIONS ···
            END
        END
```

Figure 6.4: Implementation of Three Duelling Cowboys

	$n = 3$	$s = X \land n = 1$	$s = Y \land n = 1$	$s = Z \land n = 1$
$t = X$	$\frac{81}{140}$	1		
$t = Y$	$\frac{83}{140}$		0	
$t = Z$	$\frac{27}{70}$			0

Table 6.1: Tabulation of the probabilities for Three Duelling Cowboys

In order to prove the program in Fig. 6.4 on the preceding page is a valid implementation of the program in Fig. 6.3 on page 163, we apply the fundamental theorem and prove that the program in Fig. 6.4 on the preceding page refines the following programs:

$$\frac{81}{140} \mid s : \langle s = X \rangle \tag{6.27}$$

$$\frac{27}{320} \mid s : \langle s = Y \rangle \tag{6.28}$$

$$\frac{151}{448} \mid s : \langle s = Z \rangle \tag{6.29}$$

$$1 \mid s : 1 \,. \tag{6.30}$$

Proving the refinement for Prog. (6.30) means that the implementation terminates and makes changes to s only, which holds trivially. We concentrate on proving the refinement for Prog. (6.27), since the refinements for Prog. (6.28) and Prog. (6.29) will be similar.

We apply the rules in Sec. 5.4.2 for Prog. (6.27). We have to prove that

$$\frac{81}{140} \quad \Rrightarrow \quad [s_0 := s]\,[ThreeCowboyXYZ]\,\langle s = X \rangle \,,$$

which is equivalent to

$$\frac{81}{140} \quad \Rrightarrow \quad [ThreeCowboyXYZ]\,\langle s = X \rangle \,,$$

since there are no s_0 variables in the operation ThreeCowboyXYZ (the implementation version of the operation).

We try to "guess" the expectation of the loop using the Tab. 6.1, which summarises the probability of establishing $s = X$ after executing the body of the loop.

In the table, we leave out those states where $n = 2$, since they are invalid states during the execution of the loop[5]. We also leave out the probabilities for some

[5]In fact, they are hidden inside the sub-operations which model the duelling between pairs of cowboys.

invalid states in the table, e.g. when $t = X \wedge s = Y \wedge n = 1$. The table can be read as follows. If we are in the state where $n = 3$ and $t = X$, i.e. all cowboys are alive and it is X's turn to fire, there will be another iteration of the loop, and the probability that $s = X$ afterwards is $\frac{81}{140}$ (according the informal reasoning). Similarly, we have other probabilities when $n = 3 \wedge t = Y$ and $n = 3 \wedge t = Z$. Moreover, if we are in the state where $n = 1 \wedge s = X \wedge t = X$, the loop terminates and the probability of establishing $s = X$ is 1. Clearly, that probability is 0 when $s = Y$ or $s = Z$. Putting all these together, we have the guessed expectation for the loop is

$$E_2 \;\widehat{=}\; \begin{array}{lll} \frac{81}{140} & \times & \langle n = 3 \wedge t = X \rangle \quad + \\ \frac{83}{140} & \times & \langle n = 3 \wedge t = Y \rangle \quad + \\ \frac{27}{70} & \times & \langle n = 3 \wedge t = Z \rangle \quad + \\ & & \langle s = X \wedge n = 1 \rangle \,. \end{array}$$

With respect to Sec. 5.4.2 about the partial correctness of probabilistic loop, we have further information for the *WHILE* loop in the implementation of operation ThreeCowboyXY as follows.

- The standard invariant is $I_2 \;\widehat{=}\; n \in \{1, 3\}$.
- The guard for the loop is $G_2 \;\widehat{=}\; n = 3$.
- The body of the loop is

$$S_2 \;\widehat{=}\; \mathit{IF} \cdots \mathit{ELSIF} \cdots \mathit{THEN} \cdots \mathit{ELSE} \cdots \mathit{END} \,.$$

- The post-condition Q_2 is the constant predicate $true$.
- The post-expectation is $B_2 \;\widehat{=}\; \langle s = X \rangle$.

We concentrate on proving the "maintenance" of E_2 during the execution of the loop, since it will show how the (inner) multiple probabilistic specification substitutions interact. The other proofs are similar to the previous example. We need to prove that

$$E_2 \;\Rrightarrow\; [S_2]\, E_2 \,,$$

under the assumption that $G_2 \wedge I_2$ holds. We begin the reasoning from the left-hand side as follows:

E_2

$$\equiv \quad \tfrac{81}{140} \times \langle n = 3 \wedge t = X \rangle \; + \qquad \text{definition of } E_2$$
$$\tfrac{83}{140} \times \langle n = 3 \wedge t = Y \rangle \; +$$
$$\tfrac{27}{70} \times \langle n = 3 \wedge t = Z \rangle \; +$$
$$\langle s = X \wedge n = 1 \rangle$$

$$\equiv \quad \tfrac{81}{140} \times \langle 3 = 3 \wedge t = X \rangle \; + \qquad G_2 \;\Rightarrow\; n = 3$$
$$\tfrac{83}{140} \times \langle 3 = 3 \wedge t = Y \rangle \; +$$
$$\tfrac{27}{70} \times \langle 3 = 3 \wedge t = Z \rangle \; +$$
$$\langle s = X \wedge 3 = 1 \rangle$$

$\equiv$ logic, embedded predicate and arithmetic

$$\tfrac{81}{140} \times \langle t = X \rangle + \tfrac{83}{140} \times \langle t = Y \rangle + \tfrac{27}{70} \times \langle t = Z \rangle \; .$$

For the right-hand side, the reasoning can be seen as follows:

$[S_2]\, E_2$

$\equiv$ definition of S_2 [6]

$$[t = X \implies S_{2a} \;[\!]\; t = Y \implies S_{2b} \;[\!]\; t = Z \implies S_{2c}]\, E_2$$

$\equiv$ guarded substitution

$$\begin{array}{l} \tfrac{1}{\langle t=X \rangle} \times [S_{2a}]\, E_2 \\ \min \\ \tfrac{1}{\langle t=Y \rangle} \times [S_{2b}]\, E_2 \\ \min \\ \tfrac{1}{\langle t=Z \rangle} \times [S_{2c}]\, E_2 \end{array}$$

Considering $[S_{2a}]\, E_2$, we have

[6] S_{2a}, S_{2b} and S_{2c} are corresponding probabilistic choice branches in the body of the *WHILE* loop.

$$[S_{2a}]\, E_2$$

$\equiv$ definition of S_{2a}

$$\Big[(s \longleftarrow \mathsf{TwoCowboyZX};\, n :=\, 1) \;{}_{\frac{2}{3}}\!\oplus\, t :=\, Y\Big]\, E_2$$

$\equiv$ probabilistic choice substitutions

$$\begin{array}{rcll} \frac{2}{3} & \times & [s \longleftarrow \mathsf{TwoCowboyZX};\, n :=\, 1] & E_2 \\ + & & & \\ (1-\frac{2}{3}) & \times & [t :=\, Y] & E_2 \end{array}$$

$\equiv$ details of the calculations are in Appendix D.2

$$\tfrac{2}{3} \times [s \longleftarrow \mathsf{TwoCowboyZX}]\, \langle s = X\rangle + \tfrac{83}{420}$$

$\equiv$ specification of TwoCowboyZX

$$\tfrac{2}{3} \times \left[s : \left| \begin{array}{l} \{\frac{3}{7}, \langle s = Z\rangle\} \\ \{\frac{4}{7}, \langle s = X\rangle\} \end{array} \right. \right] \langle s = X\rangle + \tfrac{83}{420}$$

$\equiv$ multiple probabilistic specification substitution

$$\tfrac{2}{3} \times \left[\left| \begin{array}{lll} (1 & \mid\; s :\, \langle s = Z\rangle) & @\, \frac{3}{7} \\ (1 & \mid\; s :\, \langle s = X\rangle) & @\, \frac{4}{7} \end{array} \right. \right] \langle s = X\rangle + \tfrac{83}{420}$$

$\equiv$ multi-way probabilistic choice substitution

$$\tfrac{2}{3} \times \left(\begin{array}{rl} & \frac{3}{7} \times [1 \mid\; s :\, \langle s = Z\rangle]\, \langle s = X\rangle \\ + & \frac{4}{7} \times [1 \mid\; s :\, \langle s = X\rangle]\, \langle s = X\rangle \end{array} \right) + \tfrac{83}{420}$$

$\equiv$ probabilistic specification substitutions

$$\tfrac{2}{3} \times \left(\begin{array}{l} \frac{3}{7} \times 1 \times [s_0 :=\, s] \sqcap s \cdot (\langle s = Z\rangle \div \langle s = X\rangle) \\ +\frac{4}{7} \times 1 \times [s_0 :=\, s] \sqcap s \cdot (\langle s = X\rangle \div \langle s = X\rangle) \end{array} \right) + \tfrac{83}{420}$$

$\equiv$ the first min is 0 when $s = X$
the second min is 1 when $s = X$

$$\tfrac{2}{3} \times \left(\begin{array}{rl} & \frac{3}{7} \times [s_0 :=\, s]\, 0 \\ + & \frac{4}{7} \times [s_0 :=\, s]\, 1 \end{array} \right) + \tfrac{83}{420}$$

$\equiv$ simple substitutions and arithmetic

$$\tfrac{81}{140}\,.$$

With similar calculations, we have

$$[S_{2b}]\, E_2 \equiv \frac{83}{140}\ ,$$

and

$$[S_{2b}]\, E_2 \equiv \frac{27}{70}\ .$$

So we have overall

$$\begin{array}{lll}
& [S_2]\, E_2 & \\
\equiv & & \text{calculation above} \\
& \left(\frac{1}{\langle t{=}X\rangle} \times \frac{81}{140}\right) \quad \min \quad \left(\frac{1}{\langle t{=}Y\rangle} \times \frac{83}{140}\right) \quad \min \quad \left(\frac{1}{\langle t{=}Z\rangle} \times \frac{27}{70}\right) & \\
\equiv & \frac{81}{140} \times \langle t = X\rangle + \frac{83}{140} \times \langle t = Y\rangle + \frac{27}{70} \times \langle t = Z\rangle & \text{arithmetic of } \min \\
\equiv & E_2\ . &
\end{array}$$

Thus we have proved that

$$E_2 \quad \Rrightarrow \quad [S_2]\, E_2\ ,$$

which means that the expectation E_2 does not decrease during the execution of the loop. With other trivial proofs, we can conclude that the implementation refines Prog. (6.27). Similarly, using different expectations for loop, we can prove that the implementation also refines Prog. (6.28) and Prog. (6.29). According to the multiple probabilistic fundamental theorem, it is valid implementation of the three duelling cowboy specification in Fig. 6.3 on page 163.

6.5 Proposed Changes to the *B-Toolkit*

The work proposed in this section has not yet been implemented in the *pB-Toolkit*. This can be regarded future work arising from this dissertation.

First of all, we need to introduce the *pAMN* construct that corresponds to the multiple probabilistic specification substitution

$$v : \left| \begin{array}{l} \{p_1, \langle Q_1\rangle\} \\ \{p_2, \langle Q_2\rangle\} \\ \cdots \\ \{p_n, \langle Q_n\rangle\} \end{array} \right. .$$

We can take similar approach as in the previous chapter by using the *PRE* and *ANY* clauses. The difference between multiple probabilistic specification substitution and (single) probabilistic specification substitution is that we need to "glue" the pre- and post-expectation pairs of the former together. We propose a solution for that by giving labels for each pair of pre- and post-expectations when writing *pAMN* version of the multiple probabilistic specification substitution. The following code

PRE
$i_1 : expectation(p_1) \wedge$
$i_2 : expectation(p_2) \wedge$
$\cdots \wedge$
$i_n : expectation(p_n)$ **THEN**
ANY v' **WHERE**
$i_1 : expectation(embedded(Q_{1[x_0, v/x, v']})) \wedge$
$i_2 : expectation(embedded(Q_{2[x_0, v/x, v']})) \wedge$
$\cdots \wedge$
$i_n : expectation(embedded(Q_{n[x_0, v/x, v']}))$
THEN
$v := v'$
END
END

corresponds to

$$v : \left| \begin{array}{l} \{p_1, \langle Q_1 \rangle\} \\ \{p_2, \langle Q_2 \rangle\} \\ \cdots \\ \{p_n, \langle Q_n \rangle\} \end{array} \right. ,$$

where i_j are the set of labels. The advantage of using *PRE* and *ANY* clauses is that the analyser can parse these expectations as normal predicates (pre- and post-condition). The labels only come into the play when generating proof obligations, which are then used to give the semantics for the multiple probabilistic specification substitution. This extension can be based on the modifications which are made to incorporate (single) probabilistic specification substitution as described in the previous chapter.

Secondly, the proof obligation generation for the refinement process will need to have some modifications that reflect the fundamental theorem as shown in Sec. 6.3. Again, this can be based on generating the obligations for each pair of pre- and

post-expectation separately, i.e. just generating the same obligations as for probabilistic specification substitution many times.

Moreover, we might have some other difficulties when the implementation contains loops. That is because, for each pre- and post-expectation, we have to use different expectations for loop. In general, our probabilistic loop will have many different expectations. We need to match those expectation with the correct pair of pre- and post-expectation, so we take the approach of labelling these expectations. The EXPECTATIONS clause in the loop will be extended to have labels corresponding to the labels in the specification, which will look like this:

EXPECTATIONS
$i_1 : E_1;$
$i_2 : E_2;$
$\ldots$
$i_n : E_n;$

In this way, the proof obligation generator can generate the obligations for each pre- and post-expectation pair separately and pick up the right expectation that should be used for loop.

Because we generate the obligations for each pair of pre- and post-expectation separately, we will repeatedly reason about loops. This results in having to prove the termination of the loop many times, proving the maintenance of the standard invariant of the loop repeatedly, and also the precondition of called operations more than once. This does not effect the correctness of the loop, and, in practice, it is unnecessary to do so. Further investigation needs to be done and this can be regarded as future work.

6.6 Conclusions and Future Work

In this chapter, we extended the concept of (single) probabilistic specification substitutions to cover systems that can only be described by multiple expectations. The work in this chapter allows us to formally specify and implement such systems. A practical method based on the fundamental theorem has been developed as the backbone for developing these systems. We choose to use the example of *Duelling Cowboys* to illustrate that the new specification and fundamental theorem have been accommodated successfully to the framework of developing systems in layers.

Building the Toolkit to support multiple probabilistic specification substitution and the corresponding fundamental theorem is regarded as future work arising from this dissertation. The modifications should be similar to those that have been done for the (single) probabilistic specification substitutions in previous chapter. Some issues related to this have been discussed in Sec. 6.5.

The example of Two Duelling Cowboys has been used in [49] where the authors discussed the correctness of probabilistic loops in a similar way to what we have done here.

The example of Three Duelling Cowboys can be varied so that it can have some non-deterministic behaviour in terms of firing sequence or the choice of target for each cowboys. This can lead to different probabilities of surviving the game, but the reasonings are similar to the example that is shown here. For example, instead of choosing to shoot at the cowboy next in the sequence, the cowboy can fire at the cowboy with higher accuracy rate, or the choice can be non-deterministic. The firing sequence can be alternated, for example, the cowboy who survives the last shot will have the chance to shoot.

Moreover, we derived the specification post-hoc, i.e. "backwards", from the actual implementation of the problem. In practice, we can start with symbolic probabilities in the specification. By reasoning about proof obligations of the implementation, we can drive the relationship and the actual values for those symbolic probabilities in the specification.

For loops, we can find the expectation (probabilistic invariant) of loops by tabulating the probabilities for different states. This technique can be used when state base is finite and usually simple.

In this chapter, we have the same frame for all pair of pre- and post-expectations. In the case where the frames are different, we can "standardise" for them to have the same frame. Notice that the following specifications are always the same:

$$p \mid v : \langle Q \rangle$$

and

$$p \mid (v, w) : \langle post \wedge w = w_0 \rangle \,,$$

where v, w are disjoint subsets of state variables. So for every pair of pre- and post-expectations in a multiple probabilistic specification substitution, we can use the above technique to have the same frame for all of them.

The question of completeness for probabilistic specification substitution needs to be further investigated. Here, the multiple probabilistic specification substitution only deals with systems where the post-expectations are standard and disjoint. Especially when post-expectations are numerical expressions, the intuitive meaning of the systems with such set of post-expectations is not clear, and this is regarded as a possible direction for further research.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This dissertation discusses the extension of the standard *B-Method* into the probabilistic domain. The backbone of the new methods is *pGSL*—an extension of the standard *GSL*— in which expectations (functions from state to real) replace predicates (functions from state to Boolean) in the original semantics.

In Chap. 3, we proposed a method for proving systems with probability-one termination. The new method is called *qB* and supports developments from specifications to implementations, in which the implementations can be proved to terminate with probability-one (almost-certain to terminate). The reasoning in *qB* is still based on Boolean logic, which is the main advantage of this work since most of the theoretical basis of standard *B* can be reused. This method can be applied to distributed systems where probability is used in order to break the symmetry effectively. We provided the basis for consistently developing systems of this kind from specifications to implementations. This will guarantee the final generated code is correct (with respect to probability-one correctness).

In Chap. 4, the work is taken further in order to accommodate full probabilistic reasoning (not just probability-one termination). We proposed the new concept of probabilistic machines with probabilistic invariants. We gave the intuitive meaning of the new concept, specified the rule for generating proof obligations to maintain probabilistic invariants and highlighted some unexpected results which are the main differences between standard and probabilistic domains.

In Chap. 5, the extension is widened to cover the concept of stepwise refinement. We introduced probabilistic specification substitutions and the fundamental theorem to refine these specifications to code. We separated the standard and probabilistic reasoning so that most of the Boolean reasoning in the standard *B-Toolkit* could be reused. The extra probabilistic reasoning is supported by extending the library of proof rules to deal with real numbers. We also investigated some subtle issues that occurred, and introduced the terminating version of probabilistic specification substitutions to tackle these problems. Using the terminating probabilistic specification substitution allows us to develop systems in layers, which is one of the key techniques within the *B-Method*.

In Chap. 6, we extended probabilistic specification substitutions for systems with multiple expectations. The fundamental theorem is also developed for the new substitutions, in order to refine such specifications to code.

In conclusion, with those methods that we proposed in this dissertation, systems with probabilistic properties can be developed formally from specifications to implementations. We also modified the *B-Toolkit* to support the work. The modified *B-Toolkit* is used in order to assist with generating and proving obligations about the correctness of systems that are developed within the new methods.

7.2 Future work

In this dissertation, we provide the framework for probabilistic *B-Method* which can be used to develop probabilistic systems formally. Moreover, we concentrated on the practical purpose of building a supporting toolkit. This section suggests some future directions.

7.2.1 Tool Support for Multiple Expectation Systems

The *B-Toolkit* can be modified to support the multiple probabilistic specification substitution and the corresponding fundamental theorem. This includes the support for syntax and semantics of new constructs, and proof obligation generation to maintain the correctness of systems which used these constructs.

The main issues related to this can be seen in Sec. 6.5. This work can be based on those modifications which have been already done for (single) probabilistic specification substitutions.

7.2.2 Completeness of Probabilistic Specification Substitutions

The probabilistic specification substitution introduced in Chap. 5 is incomplete. We proposed the multiple probabilistic specification substitution in order to be able to (abstractly) specify more probabilistic systems based on a set of pre- and post-expectations that they conform to. The question is "Is the multiple probabilistic specification substitution complete or not"? Can the multiple probabilistic specification substitution be use to describe every programs in *pGSL*? If it is incomplete, what is the set of programs that the multiple probabilistic specification substitution covers? By answering these questions can give us a better understanding of the scope of our method, i.e. which problems we can or cannot model.

7.2.3 Composition of Probabilistic Machines

In *B*, machines can be composed to model larger system. The meaning of the large machine is given by the meaning of the sub-machines and the way they are composed. With our proposal of probabilistic invariants for probabilistic machines in Chap. 4, probabilistic properties of systems can be described by the new kind of invariants. However, the meaning of these invariants when machines are composed needs to be investigated further.

7.2.4 Data Refinement

The refinement described in Chap. 5 and Chap. 6 is algorithmic refinement, i.e. they are not the general data refinement which is usually used in *B* (algorithmic refinement is a special case of data refinement).

In general, the data refinement rule (for both standard and probabilistic context) is described in second-order logic. Fortunately, in the case of the standard context, there is an equivalent form of the data refinement rule in first-order logic [20]; it is called Gries' rule. This is the main reason why the *B-Toolkit* can generate and discharge proof obligations in first-order logic.

In the probabilistic context, there are no equivalent first-order rules for data refinement (in general). The reason for that is the lack of conjunctivity of probabilistic programs. The conjunctivity is replaced by the sub-conjunctivity property [49]. Still, this is not enough to guarantee an equivalent form of data refinement

rule in first-order logic. The paper by Schneider et. al. [61] discusses the refinement for a case study in *pAMN*, but the reasoning remains purely at the program level, not at the expectation level.

The lack of a first-order refinement rule will prevent the integration of probabilistic data refinement into *pB*, which is supported by the (first-order) *B-Toolkit*, in general. However, in some special cases, we can establish an equivalent first-order rule. In these cases, we can separate the probabilistic and standard constructs and reason about them separately. The data refinement representing the standard context can be reasoned about using Gries' rule. The probabilistic construct keeps the structural (algorithmic) refinement but within a small state base and the probabilistic data refinement can be proved by considering all the possible states. This will be a possible future direction for integrating probabilistic data refinement into *pB*.

Moreover, we have introduced probabilistic choice substitutions which can appear in the specification. However, how to refine the probabilistic choice substitution in a way which is consistent with the data refinement concept in *pB* remains as future work.

7.2.5 Probabilistic *Event-B*

With the extension from (classical) *B* to *Event-B*, naturally, there is a possibility for extending *Event-B* to "probabilistic *Event-B*". The paper by Morgan et. al. [46] discusses about this possibility and some (early) challenges for having probability within *Event-B*.

The key for this work will be the balance between theoretical reasoning and having a practical supporting tool. Treating the full reasoning for probability, with abstraction and refinement, is complicated, and usually unnecessary in practice. The solution seems to depend on the capability of separating (as much as possible) the standard and probabilistic constructs and reasoning about them separately.

7.2.6 Animation

One of the advantages of the *B-Toolkit* over other *B*'s supporting tools is the animator. The animator is used to verify specifications against user's requirements. It is important to have the specification is consistent with the user's requirement, since the specification is the first step for formal development [31, 36]. It is also

true in *pB* that we need to be confident about the correctness of the specification.

Verifying a probabilistic specification is not the same as in the standard context, where the correctness can be checked at every state. What does it mean to have a "correct" probabilistic substitution? For every probabilistic system, you need to have "enough" tests in order to do any meaningful statistical reasoning for this. Moreover, for our probabilistic systems, we need to "execute" the systems by calling a sequence of operations. There will be questions on how the sequence is chosen (on-the-fly or before). Should we "re-run" this sequence repeatedly in order to have some distribution of outputs, or how long the sequence should be in order to have real meaningful probabilistic effects? Furthermore, this seems to be related to confidence intervals in probability theory, e.g. we only can say something similar to "with the confidence level of 95%, the system is correct".

Another alternative to animation is model checking. Some tools have been built to model check *B* specifications, notably the ProB [35, 36, 34]. Extension to ProB will be a possible direction for future research. There are even existing tools for model-checking probabilistic systems such as PRISM [55]. But the scope of PRISM is for more general systems in the form of Markov chains and Markov decision processes.

Bibliography

[1] Jean-Raymond Abrial. *B-Tool Reference Manual*.

[2] Jean-Raymond Abrial. *B-Tool Tutorial*.

[3] Jean-Raymond Abrial. Extending B without Changing It (for Developing Distributed Systems). In Habrias [21], pages 169–191.

[4] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.

[5] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing*, 14(3):215–227, 2003.

[6] Jean-Raymond Abrial and Louis Mussat. Introducing Dynamic Constraints in B. In Bert [10], pages 83–128.

[7] Ralph-Johan Back. *On the Correctness of Refinement in Program Development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.

[8] Ralph-Johan Back and Joakim Von Wright. *Refinement Calculus, a Systematic Itroduction*. Springer-Verlag New York, Inc., 1998.

[9] Richard Banach and Simon Fraser. Retrenchment and the B-Toolkit. In Treharne et al. [66].

[10] Didier Bert, editor. *B'98: Proceedings of the 2nd International B Conference*, volume 1393 of *Lecture Notes in Computer Science*, Montpellier, France, April 1998. Springer Verlag.

[11] Didier Bert, Jonathan Bowen, Steve King, and Marina Waldén, editors. *ZB2003: Formal Specification and Development in Z and B, Proceedings of the 3rd International Conference of B and Z Users*, volume 2651 of *Lecture Notes in Computer Science*, Turku, Finland, June 2003. Springer Verlag.

[12] Tatibouet Bruno. jBTools. http://lifc.univ-fcomte.fr/~tatibouet/JBTOOLS/index_en.html.

[13] Tatibouet Bruno. StudioB. http://lifc.univ-fcomte.fr/~tatibouet/EDITEURB/index.html.

[14] ClearSy. Atelier B. http://www.atelierb.societe.com/index_uk.htm.

[15] ClearSy. B4free. http://www.B4free.com.

[16] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[17] Colin Fidge and Carron Shankland. But What if I Don't Want to Wait Forever? *Formal Aspects of Computing*, 14(3):281–294, 2003.

[18] John E. Freund. *John E. Freund's Mathematical Statistics*. Prentice Hall International, Inc., 6 edition, 1999.

[19] David Gries. A Note on a Standard Strategy for Developing Loop Invariants and Loops. *Science of Computer Programming*, 2:207–214, 1984.

[20] David Gries and Jan Prins. A New Notion of Encapsulation. In *Symposium on Language Issues in Programming Environments*. SIGPLAN, June 1985.

[21] Henri Habrias, editor. *Proceedings of the 1st Conference on the B method*, Putting into Practice Methods and Tools for Information System Design, Nantes, France, November 1996. IRIN Institut de recherche en informatique de Nantes.

[22] Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of Probabilistic Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):356–380, 1983.

[23] Thai Son Hoang, Zhendong Jin, Ken Robinson, Annabelle McIver, and Carroll Morgan. Probabilistic Invariants for Probabilistic Machines. In Bert et al. [11], pages 240–259.

[24] Thai Son Hoang, Zhendong Jin, Ken Robinson, Annabelle McIver, and Carroll Morgan. Proofs of the Min-Cut Development. http://www.cse.unsw.edu.au/~htson/publications/minCutProofs.pdf, April 2004.

[25] Thai Son Hoang, Zhendong Jin, Ken Robinson, Annabelle McIver, and Carroll Morgan. Development via Refinement in Probabilistic B — Foundation and Case Study. In Treharne et al. [66], pages 355–373.

[26] C.A.R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–583, October 1969.

[27] IEEE. *IEEE Standard for a High Performance Serial Bus. Std 1394-1995*, 1995.

[28] IEEE. *IEEE Standard for a High Performance Serial Bus (supplement). Std 1394a-2000*, 2000.

[29] INRIA. The Coq Proof Assistant. http://coq.inria.fr/.

[30] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, Inc., 2nd edition, 1990. ISBN 0-13-880733-7.

[31] Cliff Jones. Specification Before Satisfaction: The Case for Research into Obtaining the Right Specification. In Treharne et al. [66], pages 1–5.

[32] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development*. Formal Approaches to Computing and Information Technology. Springer Verlag, 1996.

[33] Kevin Lano and Howard Haughton. *Specification in B: An Introduction Using the B-Toolkit*. Imperial College Press, 1996.

[34] Michael Leuschel. ProB. http://www.ecs.soton.ac.uk/~mal/systems/prob.html.

[35] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, page 8550874. Springer Verlag, 2003.

[36] Michael Leuschel and Edd Turner. Visualising Larger State Spaces in ProB. In Treharne et al. [66], pages 6–23.

[37] B-Core(UK) Ltd. The B-Toolkit. http://www.b-core.com.

[38] B-Core(UK) Ltd. *B-Toolkit User's Manual*, 1997.

[39] Annabelle McIver. Quantitative Program Logic and Counting Rounds in Probabilistic Distributed Algorithms. In *Proceedings of the 5th International Workshop ARTS '99*, volume 1601 of *Lecture Notes in Computer Science*, 1999.

[40] Annabelle McIver and Carroll Morgan. Demonic, Angelic and Unbounded Probabilistic Choice in Sequential Programs. *Acta Informatica*, 37:329–354, 2001.

[41] Annabelle McIver, Carroll Morgan, and Thai Son Hoang. Probabilistic termination in B. In Bert et al. [11], pages 216–239.

[42] Carroll Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [51].

[43] Carroll Morgan. *Programming from Specifications*. Prentice Hall International, Inc., second edition, 1994. At http://web.comlab.ox.ac.uk/oucl/publications/books/PfS.

[44] Carroll Morgan. Proof Rules for Probabilistic Loops. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, July 1996. At http://www.springer.co.uk/ewic/workshops/7RW; also available at [56, M95].

[45] Carroll Morgan. The Generalised Substitution Language Extended to Probabilistic Programs. In *Proceedings B'98: the 2nd International B Conference*,

volume 1393 of *Lecture Notes in Computer Science*, Montpelier, April 1998. Also available at [56, B98].

[46] Carroll Morgan, Thai Son Hoang, and Jean-Raymond Abrial. The Challenge of Probabilistic Event-B. In Treharne et al. [66], pages 162–171.

[47] Carroll Morgan and Annabelle McIver. An Expectation-Based Model for Probabilistic Temporal Logic. *Logic Journal of the IGPL*, 7(6):779–804, 1999. Also available at [56, MM97].

[48] Carroll Morgan and Annabelle McIver. Almost-certain Eventualities and Abstract Probabilities in the Quantitative Temporal Logic *qTL*. In *Proceedings CATS '01*. Elsevier, 2000. Also available at [56, PROB-1]; to appear in *Theoretical Computer Science*.

[49] Carroll Morgan and Annabelle McIver. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer Verlag, 2005.

[50] Carroll Morgan and Ken Robinson. *On the Refinement Calculus*, chapter Specification Statements and Refinements, pages 23–45. Springer Verlag, 1992.

[51] Carroll Morgan and Trevor Vickers, editors. *On the Refinment Calculus*. FACIT Series in Computer Science. Springer Verlag, Berlin, 1994.

[52] Joseph Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[53] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[54] Amir Pnueli. On the Extremely Fair Treatment of Probabilistic Algorithms. In *Proceedings of the 15th Annual Symposium on the Theory of Computing*, pages 278–290, 1983.

[55] PRISM. Probabilistic Symbolic Model Checker. http://www.cs.bham.ac.uk/~dxp/prism/publications.html.

[56] PSG. Probabilistic Systems Group: Collected Reports. At http://web.comlab.ox.ac.uk/oucl/research/areas/probs/bibliography.html.

[57] Michael Rabin. The Choice Coordination Problem. *Acta Informatica*, 17:121–134, 1982.

[58] Josyula R. Rao. *Building on the UNITY Experience: Compositionality, Fairness and Probability in Parallelism*. PhD thesis, University of Texas at Austin, 1992.

[59] Josyula R. Rao. Reasoning about Probabilistic Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[60] Steve Schneider. *The B-Method: An Introduction*. Cornerstones of Computing. Palgrave Publishers Ltd., 2001.

[61] Steve Schneider, Thai Son Hoang, Ken Robinson, and Helen Treharne. Tank Monitoring: A pAMN Case Study. Electronic Notes in Theoretical Computer Science (ENTCS), April 2005.

[62] David Simons and Mariëlle Stoelinga. Mechanical Verification of the IEEE 1394a Root Contention Protocol using Uppaal2k. *Springer International Journal of Software Tools for Technology Transfer*, pages 469–485, 2001.

[63] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, Inc., 2nd edition, 1992.

[64] Mariëlle Stoelinga. Fun with FireWire: A Comparative Study of Formal Verification Methods Applied to the IEEE 1394 Root Contention Protocol. *Formal Aspects of Computing*, 14(3):328–337, 2003.

[65] Mariëlle Stoelinga and Frits Vaandrager. Root Contention in IEEE 1394. In *Proceedings of the 5th AMAST workshop on real time and probabilistic systems Bamberg, Germany, ARTS' 1999*, volume 1061 of *Lecture Notes in Computer Science*, pages 53–74. Springer Verlag, 1999.

[66] Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors. *ZB2005: Formal Specification and Development in Z and B, Proceedings of*

the 4th International Conference of B and Z Users, volume 3455 of *Lecture Notes in Computer Science*, Guildford, United Kingdom, April 2005. Springer Verlag.

[67] Neil White. Probabilistic Specification and Refinement. Master's thesis, Keble College, Oxford, September 1996.

[68] John Wordsworth. *Software Engineering with B*. Addison Wesley Longman Ltd., 1996.

[69] Lenore D. Zuck. *Past Temporal Logic*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, 1986.

[70] Lenore D. Zuck, Amir Pnueli, and Yonit Kesten. Automatic Verification of Probabilistic Free Choice. In *VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 208–224. Springer Verlag, 2002.

Appendix A

Root Contention

A.1 Specification: FirewireResolve.mch

MACHINE *FirewireResolve*
SETS
  $STATUS = \{ short\_signal , long\_signal \}$

OPERATIONS
  $xx , yy \longleftarrow$ **Resolve** $\widehat{=}$
    CHOICE
      $xx := short\_signal \parallel yy := long\_signal$
    OR
      $xx := long\_signal \parallel yy := short\_signal$
    END
END

A.2 Implementation: FirewireResolveI.imp

IMPLEMENTATION *FirewireResolveI*
REFINES *FirewireResolve*
SEES
Firewire_AbstractChoice , *Bool_TYPE*

OPERATIONS
xx , *yy* ⟵ **Resolve** $\widehat{=}$
VAR *tempx* , *tempy* , *bb* **IN**
tempx := *short_signal* **;**
tempy := *short_signal* **;**
WHILE *tempx* = *tempy* **DO**
bb ⟵ *Firewire_AbstractChoice* **;**
IF *bb* = *TRUE* **THEN**
tempx := *short_signal*
ELSE
tempx := *long_signal*
END ;
bb ⟵ *Firewire_AbstractChoice* **;**
IF *bb* = *TRUE* **THEN**
tempy := *short_signal*
ELSE
tempy := *long_signal*
END BOUND *1*
VARIANT *prob* (*tempx* = *tempy*)
INVARIANT *tempx* ∈ *STATUS* ∧ *tempy* ∈ *STATUS*
END ;
xx := *tempx* **;** *yy* := *tempy*
END
END

Appendix B

Rabin's Choice Coordination Algorithm

B.1 Finite Bag: FBag.mch

MACHINE *FBag*

SEES

FBag_ctx machine contain the general definitions about bags.

FBag_ctx

VARIABLES

State contains one variable.

bag

INVARIANT

bag is modelled as partial function from natural number to natural number (definition of *Bag*). The domain of *bag* is a finite set. Each element in the bag has an index mapped to.

$bag \in Bag \wedge$

$\mathsf{dom}\ (\ bag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ )$

INITIALISATION

Initially, the *bag* is empty.

$bag := \{\}$

OPERATIONS

Operation: SetToBag
Requirements: Set the contents of *bag* to input *bb*.
Precondition: bb is a finite bag.

SetToBag (*bb*) $\widehat{=}$

PRE

$bb \in Bag \wedge$

$\mathsf{dom}\ (\ bb\ ) \in \mathbb{F}\ (\ \mathbb{N}\ )$

THEN

$bag := bb$

END ;

Operation: Takelem
Requirements: Take one element *ee* from the *bag*. By finding the index for element (using *ANY* clause). Remove the mapping from *bag*.
Precondition: The element *ee* is in the *bag*.

Takelem (*ee*) $\widehat{=}$

PRE

$ee \in \mathsf{ran}\ (\ bag\ )$

THEN

ANY *nn*

WHERE

$nn \in \mathsf{dom}\ (\ bag\ ) \wedge bag\ (\ nn\ ) = ee$

THEN

$bag := \{\ nn\ \} \lhd bag$

END

END ;

Operation: Addelem

Requirements: Adding element *ee* to the *bag*. Choose a free index *nn*, and adding the element to the *bag* according to new index.

Precondition: The element *ee* is a natural number.

Addelem (*ee*) $\widehat{=}$

PRE

$ee \in \mathbb{N}$

THEN

ANY *nn*

WHERE

$nn \in \mathbb{N} \wedge nn \notin \mathsf{dom}\ (\ bag\ )$

THEN

$bag := bag \lhd\!\!\!- \{\ nn \mapsto ee\ \}$

END

END ;

Operation: Anyelem

Requirements: Choose any element from the *bag*.

Precondition: The *bag* is not empty.

ee $\longleftarrow$ **Anyelem** $\widehat{=}$

PRE

$bagSize\ (\ bag\ ) \neq 0$

THEN

$ee :\in \mathsf{ran}\ (\ bag\ )$

END ;

Operation: Size

Requirements: Return the size of the bag.

Precondition: There are no preconditions.

$ss \longleftarrow$ **Size** $\widehat{=}$

$ss := bagSize\ (\ bag\ )$

END

B.2 Context of Finite Bag: FBag_ctx.mch

MACHINE *FBag_ctx*

SEES

Getting the general math information from Math machine.

Math

DEFINITIONS

Bag is defined as a partial function from natural number (indexes) to natural number (element of the bag).

$Bag \;\widehat{=}\; \mathbb{N} \pfun \mathbb{N}$ **;**

The size of a bag is the cardinality.

$bagSize(b) \;\widehat{=}\; \mathsf{card}(b)$ **;**

maximum element in the bag which is the maximum number of the range of the bag or 0 if the bag is empty.

$maxInBag(b) \;\widehat{=}\; max0(\mathsf{ran}(b))$ **;**

Define the number of elements in a bag b that greater than n.

$bagGreat(b, n) \;\widehat{=}\; \mathsf{card}(b \rhd \{ xx \mid xx \in \mathbb{N} \land xx \geq n \})$

END

B.3 Specification: Rabin.mch

Simple abstract specification of the Rabin Choice Coordination problem. The parameter *maxtotal* is the maximum total number allowed. The constraint on it is that *maxtotal* is less than the max integer allowed in *B*.

MACHINE *Rabin* (*maxtotal*)
CONSTRAINTS
$maxtotal \leq MaxScalar$
SEES
Scalar_TYPE

OPERATIONS

Operation: Decide

Requirements: Given two number *lout* and *rout*, which corresponding to the number of people outside the two places *left* and *right* respectively. The operation arranges those set of people to be either inside the *right* or *left* (*lin* or *rin*)

Precondition: The total number of the people is not over the *maxtotal*.

lin , $rin \longleftarrow$ **Decide** ($lout$, $rout$) $\widehat{=}$
PRE
$lout \in \mathbb{N} \wedge rout \in \mathbb{N} \wedge$
$lout + rout \leq maxtotal$
THEN
CHOICE
$lin := lout + rout \ \|$
$rin := 0$
OR
$rin := lout + rout \ \|$
$lin := 0$
END
END
END

B.4 Refinement: RabinR.ref

Refinement of the Choice Coordination problem using the definition of bags.

REFINEMENT *RabinR*

REFINES *Rabin*

SEES

Getting the global knowledge about bags.

FBag_ctx

INCLUDES

Including for instances of bag to represent four sets of people: inside the *left* (*lin*), inside the *right* (*rin*), outside the *left* (*lout*) and outside the *right* (*rout*).

lin . FBag , *rin . FBag* , *lout . FBag* , *rout . FBag*

OPERATIONS

Operation: Decide

Requirements: Refinement of operation Decide. Here, the number is represented by the cardinality of the bags. Using the non-deterministic clause to choose the final value of the four bags accordingly: There will be no one outside the two places (*floutbag* and *froutbag* are empty) and either all people end up on the *left* or on the *right* inside.

The value of the output will be the size of the bags after having the final values.

lin , *rin* $\longleftarrow$ **Decide** (*lout* , *rout*) $\widehat{=}$

BEGIN

ANY *flinbag* , *frinbag* , *floutbag* , *froutbag*

WHERE

$flinbag \in Bag \wedge frinbag \in Bag \wedge floutbag \in Bag \wedge froutbag \in Bag \wedge$

$\mathsf{dom}\ (\ flinbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge \mathsf{dom}\ (\ frinbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge$

$\mathsf{dom}\ (\ floutbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge \mathsf{dom}\ (\ froutbag\ ) \in \mathbb{F}\ (\ \mathbb{N}\ ) \wedge$

$floutbag = \{\} \wedge froutbag = \{\} \wedge$

$(\ bagSize\ (\ flinbag\ ) = 0 \vee bagSize\ (\ frinbag\ ) = 0\ ) \wedge$

bagSize (*flinbag*) + *bagSize* (*frinbag*) = *lout* + *rout*

THEN

lin . *SetToBag* (*flinbag*) || *rin* . *SetToBag* (*frinbag*) ||

lout . *SetToBag* (*floutbag*) || *rout* . *SetToBag* (*froutbag*)

END ;

lin ⟵ *lin* . *Size* ||

rin ⟵ *rin* . *Size*

END

END

B.5 Implementation: RabinRI.imp

Implementation of the Rabin's solution to the Choice Coordination problem.

IMPLEMENTATION *RabinRI*
REFINES *RabinR*
SEES

Seeing the information about Boolean, bags and some maths definitions.

Bool_TYPE , *FBag_ctx* , *Math* , *tourist_AbstractChoice*

Importing a machine that contains the state and a step change of the algorithm.

IMPORTS *RabinChoice* (*maxtotal*)

OPERATIONS

Operation: Decide
Requirements: Implementation of operation Decide. Setup the basic four bags (represent four sets of people accordingly). While there are some people outside then continue update the pad of those people until all of them are inside the two places.

lin , *rin* ⟵ **Decide** (*lout* , *rout*) $\widehat{=}$

VAR

sizelout , *sizerout* , *bb*

IN

InitState (*lout* , *rout*) **;**

sizelout ⟵ *loutSize* **;**

sizerout ⟵ *routSize* **;**

WHILE

$sizelout \neq 0 \vee sizerout \neq 0$

DO

bb ⟵ *tourist_AbstractChoice* **;**

UpdatePad (*bb*) **;**

sizelout ⟵ *loutSize* **;**

sizerout ⟵ *routSize* **BOUND** $9 \times total$

VARIANT

In here the variant is decrease according to the 0-1 law of termination.

$rEqual(LL, RR) \times 3 \times total + 3 \times total - ($

$3 \times (bagSize(linbag) + bagSize(rinbag)) +$

$(bagGreat(loutbag, LL) + bagGreat(routbag, LL)) +$

$(bagGreat(loutbag, RR) + bagGreat(routbag, RR)))$

INVARIANT

$bagSize(loutbag) = sizelout \wedge$

$bagSize(routbag) = sizerout \wedge$

$total = lout + rout$

END ;

Getting the size of the inside bags after the arrangement.

lin ⟵ *linSize* **;**

rin ⟵ *rinSize*

END

END

B.6 Support the Implementation: RabinState.mch

Specification of the algorithm state. There is one parameter to specify the maximum number of people allowed.

MACHINE *RabinState* (*maxtotal*)

CONSTRAINTS

$maxtotal \leq MaxScalar$

SEES

Getting the global information about bags and maths definitions.

Math , *FBag_ctx* , *Scalar_TYPE*

INCLUDES

This machine contains four bags representing four sets of people.

lin . FBag , *rin . FBag* , *lout . FBag* , *rout . FBag*

PROMOTES

Promoting the following operations from those included machines

lin . Size , *rin . Size* , *lout . Size* , *rout . Size* , *lout . Anyelem* , *rout . Anyelem*

VARIABLES

Additional state to the algorithm (in addition to the states of included machines): *total* is the total number of people, *LL* and *RR* are the numbers appearing on the board outside the two places *left* and *right*, respectively.

total , *LL* , *RR*

INVARIANT

total is always equal to the sum of the cardinalities of 4 bags.

$total \in \mathbb{N} \wedge$

$total = bagSize\ (\ linbag\ ) + bagSize\ (\ rinbag\ ) +$

$(\ bagSize\ (\ loutbag\ ) + bagSize\ (\ routbag\ )\ ) \wedge$

Type of *LL* and *RR*

$LL \in \mathbb{N} \wedge RR \in \mathbb{N} \wedge$

The following invariant is necessary to prove the correctness of the algorithm. The number on a tourist's pad never exceeds the number posted at the other place.

$maxInBag(linbag) \leq RR \wedge maxInBag(loutbag) \leq RR \wedge$

$maxInBag(rinbag) \leq LL \wedge maxInBag(routbag) \leq LL \wedge$

If any tourist has gone inside, then at least one of the tourist inside there must have a number exceeding the number posted outside.

$(bagSize(linbag) \neq 0 \Rightarrow maxInBag(linbag) > LL) \wedge$

$(bagSize(rinbag) \neq 0 \Rightarrow maxInBag(rinbag) > RR) \wedge$

This invariant is trivial and is used to prove the total correctiness of the algorithm.

$3 \times total \geq$

$3 \times (bagSize(linbag) + bagSize(rinbag)) +$

$(bagGreat(loutbag, LL) + bagGreat(routbag, LL)) +$

$(bagGreat(loutbag, RR) + bagGreat(routbag, RR))$

INITIALISATION

Initially, there are no ones and the number posted on both place is 0.

$total := 0 \parallel LL, RR := 0, 0$

OPERATIONS

Operation: InitState

Requirements: Initialise the algorithm. There are no ones inside, all the people outside have number 0 on their pad. Total number of people is the total number of people outside.

Precondition: Total number of people does not exceed *maxtotal*.

InitState $(lout, rout) \mathrel{\widehat{=}}$

PRE $lout \in \mathbb{N} \wedge rout \in \mathbb{N} \wedge lout + rout \leq maxtotal$ **THEN**

$lin.SetToBag(\{\}) \parallel$

$rin.SetToBag(\{\}) \parallel$

$lout.SetToBag((1..lout) \times \{0\}) \parallel$

$rout.SetToBag((1..rout) \times \{0\}) \parallel$

$total := lout + rout \parallel$

$LL := 0 \parallel$

$RR := 0$

END ;

Operation: MoveInLeft

Requirements: Move a person inside the *left* place with number *ll* on his pad. Modify the bag *rout* and *rin* respectively.

Precondition: There is actually a person with number *ll* on the left and number *ll* is greater than the number posted outside *left* place.

MoveInLeft (ll) $\widehat{=}$

PRE

$ll \in \mathsf{ran}\ (\ loutbag\ ) \wedge (\ bagSize\ (\ linbag\ ) \neq 0 \vee ll > LL\ )$

THEN

$lout\ .\ Takelem\ (\ ll\ )\ \|$

$lin\ .\ Addelem\ (\ ll\ )$

END ;

Operation: MoveInRight

Requirements: Move a person inside the *right* place with number *rr* on his pad. Modify the bag *lout* and *lin* respectively.

Precondition: There is actually a person with number *rr* on the right and number *rr* is greater than the number posted outside *left* place.

MoveInRight (rr) $\widehat{=}$

PRE

$rr \in \mathsf{ran}\ (\ routbag\ ) \wedge (\ bagSize\ (\ rinbag\ ) \neq 0 \vee rr > RR\ )$

THEN

$rout\ .\ Takelem\ (\ rr\ )\ \|$

$rin\ .\ Addelem\ (\ rr\ )$

END ;

Operation: MoveToLeft

Requirements: Move a person with number *rr* on his pad from the *right* to the *left* and change his number to *mm*. The number posted outside the right is changed to *mm* accordingly.

Precondition: There is actually a person with number *rr* on the right and number *mm* is greater than the number posted outside *right* place. There are no ones inside the *right* place.

MoveToLeft (rr , mm) $\widehat{=}$

PRE

$rr \in \mathsf{ran}\ (\ routbag\ ) \wedge mm \in \mathbb{N}_1 \wedge bagSize\ (\ rinbag\ ) = 0 \wedge RR \leq mm$

THEN

$rout\ .\ Takelem\ (\ rr\ )\ \|$

$lout\ .\ Addelem\ (\ mm\ )\ \|$

$RR := mm$

END ;

Operation: MoveToRight

Requirements: Move a person with number *ll* on his pad from the *left* to the *right* and change his number to *mm*. The number posted outside the left is changed to *mm* accordingly.

Precondition: There is actually a person with number *ll* on the left and number *mm* is greater than the number posted outside *left* place. There are no ones inside the *left* place.

MoveToRight (ll , mm) $\widehat{=}$

PRE

$ll \in \mathsf{ran}\ (\ loutbag\ ) \wedge mm \in \mathbb{N}_1 \wedge bagSize\ (\ linbag\ ) = 0 \wedge LL \leq mm$

THEN

$lout\ .\ Takelem\ (\ ll\ )\ \|$

$rout\ .\ Addelem\ (\ mm\ )\ \|$

$LL := mm$

END ;

Operation: LLVal

Requirements: Return the value of the number posted outside the *left*.

Precondition: There are no preconditions.

$ll \longleftarrow$ **LLVal** $\widehat{=}$

$ll := LL$ **;**

Operation: RRVal

Requirements: Return the value of the number posted outside the *right*.

Precondition: There are no preconditions.

$rr \longleftarrow$ **RRVal** $\widehat{=}$

$rr := RR$

END

Appendix C

Proof of Restricted Terminating Fundamental Theorem

We restate the fundamental theorem Thm. 7 in Sec. 5.8 here:

> Let p be an expression over x and let Q be a predicate defined over x_0, v, and satisfying $\forall\, x_0 \cdot (\exists\, v \cdot Q)$; and let T be a program written in *pGSL*. For all such programs T, if
>
> $$1 \mid v : 1 \quad \sqsubseteq \quad T \qquad \text{i.e. } T \text{ terminates and changes only variables in } v$$
>
> and
>
> $$p \mid v : \langle Q \rangle \quad \sqsubseteq \quad T$$
>
> then
>
> $$\{\!\{ p \mid v : \langle Q \rangle \}\!\} \quad \sqsubseteq \quad T\,.$$

Proof. Let E be arbitrary expectation over x. For any x_0, we have from arithmetic that

$$E \quad \Leftarrow \quad \begin{array}{l} \sqcap x \cdot (E \div \langle w = w_0 \rangle) \times \langle w = w_0 \rangle \\ (\sqcap x \cdot (E \div \langle Q^w \rangle) - \sqcap x \cdot (E \div \langle w = w_0 \rangle)) \times \langle Q^w \rangle \end{array} \quad + \tag{C.1}$$

where $Q^w \;\widehat{=}\; Q \wedge w = w_0$. We have the above inequality because the embedded predicate $\langle Q^w \rangle$ and $\langle w = w_0 \rangle$ can have value only 1 or 0. And in each case, it can be easily proved that the inequality holds, as shown below.

- If $w \neq w_0$, the right-hand side of (C.1) is 0, hence the inequality is trivially true.

- If $w = w_0 \ \wedge \ \neg Q$, the right-hand side of (C.1) is $\sqcap x \cdot (E \div \langle w = w_0 \rangle)$, which is everywhere no more than E, given that $w = w_0$.
- If $w = w_0 \ \wedge \ Q$, the right-hand side of (C.1) is $\sqcap x \cdot (E \div \langle Q^w \rangle)$ which is everywhere no more than E, given that Q^w holds.

First, we estimate the pre-expectation of T with respect to E, for all x_0 we have:

$$
\begin{array}{ll}
& [T]\, E \\
\Leftarrow & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(C.1) and monotonicity of } T \\
& [T] \left(\begin{array}{ll} \sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle) \times \langle w = w_0 \rangle & + \\ (\sqcap x \cdot (E \ \div \ \langle Q^w \rangle) - \sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle)) \times \langle Q^w \rangle & \end{array} \right) \\
\Leftarrow & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sublinearity of } T \\
& \begin{array}{ll} \sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle) \times [T] \langle w = w_0 \rangle & + \\ (\sqcap x \cdot (E \ \div \ \langle Q^w \rangle) - \sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle)) \times [T] \langle Q^w \rangle \, . & \end{array}
\end{array}
$$

Hence (since x_0 is unconstrained), in particular,

$$
\begin{array}{ll}
& [T] E \\
\Leftarrow & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_0 \text{ is given a value } x \\
& [x_0 :=\ x] \begin{pmatrix} \sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle) \times [T] \langle w = w_0 \rangle & + \\ (\sqcap x \cdot (E \ \div \ \langle Q^w \rangle) - \sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle)) \times [T] \langle Q^w \rangle & \end{pmatrix} \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{simple substitution } x_0 :=\ x \\
& [x_0 :=\ x] (\sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle) \times [T] \langle w = w_0 \rangle) \quad + \\
& \left([x_0 :=\ x] \begin{pmatrix} \sqcap x \cdot (E \div \langle Q^w \rangle) - \\ \sqcap x \cdot (E \div \langle w = w_0 \rangle) \end{pmatrix} \right) \times [x_0 :=\ x] [T] \langle Q^w \rangle \\
\equiv & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_0 \text{ is a fresh variable and simple substitution} \\
& [x_0 :=\ x] \sqcap x \cdot (E \ \div \ \langle w = w_0 \rangle) \ \times \ [x_0 :=\ x] [T] \langle w = w_0 \rangle \quad + \\
& \begin{pmatrix} [x_0 :=\ x] (\sqcap x \cdot (E \div \langle Q^w \rangle)) - \\ \sqcap x \cdot (E \div \langle w = w_0 \rangle) \end{pmatrix} \times [x_0 :=\ x] [T] \langle Q^w \rangle
\end{array}
$$

$\Leftarrow$ refinement assumption

$$[x_0 :=\ x] \sqcap x \cdot (E\ \div\ \langle w = w_0 \rangle) \times 1 \quad + \\ ([x_0 :=\ x] (\sqcap x \cdot (E\ \div\ \langle Q^w \rangle)) - \sqcap x \cdot (E\ \div\ \langle w = w_0 \rangle)) \times p$$

$\equiv$ simple substitution and arithmetic

$$[x_0 :=\ x] \sqcap x \cdot (E \div \langle w = w_0 \rangle) \times (1 - p)\ + \\ [x_0 :=\ x] (\sqcap x \cdot (E \div \langle Q^w \rangle)) \times p$$

$\equiv$ definition of probabilistic specification substitution

$$[1\ \mid\ v :\ 1]E\ \times\ (1 - p) \quad + \quad [1\ \mid\ v :\ \langle Q \rangle]E\ \times\ p$$

$\equiv$ $[(p\ \mid\ v :\ \langle Q \rangle) \quad {}_p\oplus \quad (1\ \mid\ v :\ 1)]E$ probabilistic choice substitution

$\equiv$ definition of restricted terminating probabilistic specification substitution

$$[\{\!\{p\ \mid\ v :\ \langle Q \rangle\}\!\}]E\ .$$

Because E is arbitrary, therefore $\{\!\{p\ \mid\ v :\ \langle Q \rangle\}\!\} \sqsubseteq T$.

Appendix D

Appendix for Duelling Cowboys Case Study

D.1 Two Duelling Cowboys

We have the following information:

$$S_{1a} \equiv s := X; n := 1) \; {}_{\frac{2}{3}}\oplus \; t := Y \tag{D.1}$$

$$S_{1b} \equiv s := Y; n := 1) \; {}_{\frac{1}{2}}\oplus \; t := X \tag{D.2}$$

$$E_1 \equiv \begin{array}{rl} & \langle s = X \wedge n = 1 \rangle \\ + & \langle n = 2 \wedge t = X \rangle \times \frac{4}{5} \\ + & \langle n = 2 \wedge t = Y \rangle \times \frac{2}{5} \end{array} . \tag{D.3}$$

The calculations are as follows:

$$\begin{array}{ll} & \frac{1}{\langle t = X \rangle} \times [S_{1a}]\, E_1 \\ \min & \\ & \frac{1}{\langle t \neq X \rangle} \times [S_{1b}]\, E_1 \end{array}$$

$\equiv$ Def. D.1 and Def. D.2

$$\begin{array}{ll} & \frac{1}{\langle t = X \rangle} \times \Big[(s := X; n := 1) \; {}_{\frac{2}{3}}\oplus \; t := Y\Big]\, E_1 \\ \min & \\ & \frac{1}{\langle t \neq X \rangle} \times \Big[(s := Y; n := 1) \; {}_{\frac{1}{2}}\oplus \; t := X\Big]\, E_1 \end{array}$$

$\equiv$ probabilistic choice substitutions

$$\min \begin{array}{l} \frac{1}{\langle t{=}X\rangle} \times \left(\begin{array}{lcll} & \frac{2}{3} & \times \ [(s := X; n := 1)] & E_1 \\ + & & & \\ & (1-\frac{2}{3}) & \times \ [t := Y] & E_1 \end{array} \right) \\ \frac{1}{\langle t{\neq}X\rangle} \times \left(\begin{array}{lcll} & \frac{1}{2} & \times \ [(s := Y; n := 1)] & E_1 \\ + & & & \\ & (1-\frac{1}{2}) & \times \ [t := X] & E_1 \end{array} \right) \end{array}$$

$\equiv$ Def. D.3

$$\min \begin{array}{l} \frac{1}{\langle t{=}X\rangle} \times \left(\begin{array}{ll} & \frac{2}{3} \times [(s := X; n := 1)] \\ & \left(\begin{array}{ll} & \langle s = X \wedge n = 1\rangle \\ + & \langle n = 2 \wedge t = X\rangle \times \frac{4}{5} \\ + & \langle n = 2 \wedge t = Y\rangle \times \frac{2}{5} \end{array} \right) \\ + & \\ & (1-\frac{2}{3}) \times [t := Y] \\ & \left(\begin{array}{ll} & \langle s = X \wedge n = 1\rangle \\ + & \langle n = 2 \wedge t = X\rangle \times \frac{4}{5} \\ + & \langle n = 2 \wedge t = Y\rangle \times \frac{2}{5} \end{array} \right) \end{array} \right) \\ \frac{1}{\langle t{\neq}X\rangle} \times \left(\begin{array}{ll} & \frac{1}{2} \times [(s := Y; n := 1)] \\ & \left(\begin{array}{ll} & \langle s = X \wedge n = 1\rangle \\ + & \langle n = 2 \wedge t = X\rangle \times \frac{4}{5} \\ + & \langle n = 2 \wedge t = Y\rangle \times \frac{2}{5} \end{array} \right) \\ + & \\ & (1-\frac{1}{2}) \times [t := X] \\ & \left(\begin{array}{ll} & \langle s = X \wedge n = 1\rangle \\ + & \langle n = 2 \wedge t = X\rangle \times \frac{4}{5} \\ + & \langle n = 2 \wedge t = Y\rangle \times \frac{2}{5} \end{array} \right) \end{array} \right) \end{array}$$

$\equiv$ arithmetic, sequential and simple substitutions

$$\begin{array}{l}
\frac{1}{\langle t{=}X\rangle} \times \left(\begin{array}{ll}
 & \langle X = X \wedge 1 = 1\rangle \\
\frac{2}{3}\times & + \ \langle 1 = 2 \wedge t = X\rangle \times \frac{4}{5} \\
 & + \ \langle 1 = 2 \wedge t = \ Y\rangle \times \frac{2}{5}
\end{array}\right) \\
\quad + \\
\frac{1}{\langle t{=}X\rangle} \times \left(\begin{array}{ll}
 & \langle s = X \wedge n = 1\rangle \\
\frac{1}{3}\times & + \ \langle n = 2 \wedge \ Y = X\rangle \times \frac{4}{5} \\
 & + \ \langle n = 2 \wedge \ Y = \ Y\rangle \times \frac{2}{5}
\end{array}\right)
\end{array}$$

min

$$\begin{array}{l}
\frac{1}{\langle t{\neq}X\rangle} \times \left(\begin{array}{ll}
 & \langle Y = X \wedge 1 = 1\rangle \\
\frac{1}{2}\times & + \ \langle 1 = 2 \wedge t = X\rangle \times \frac{4}{5} \\
 & + \ \langle 1 = 2 \wedge t = \ Y\rangle \times \frac{2}{5}
\end{array}\right) \\
\quad + \\
\frac{1}{\langle t{\neq}X\rangle} \times \left(\begin{array}{ll}
 & \langle s = X \wedge n = 1\rangle \\
\frac{1}{2}\times & + \ \langle n = 2 \wedge X = X\rangle \times \frac{4}{5} \\
 & + \ \langle n = 2 \wedge X = \ Y\rangle \times \frac{2}{5}
\end{array}\right)
\end{array}$$

$\equiv$ logic

$$\begin{array}{l}
\frac{1}{\langle t{=}X\rangle} \times \left(\begin{array}{l}
\frac{2}{3}\times\big(\langle \mathsf{true}\rangle + \langle \mathsf{false}\rangle \times \frac{4}{5} + \langle \mathsf{false}\rangle \times \frac{2}{5}\big) \\
+ \\
\frac{1}{3}\times\big(\langle s = X \wedge n = 1\rangle + \langle \mathsf{false}\rangle \times \frac{4}{5} + \langle n = 2\rangle \times \frac{2}{5}\big)
\end{array}\right) \\
\text{min} \\
\frac{1}{\langle t{\neq}X\rangle} \times \left(\begin{array}{l}
\frac{1}{2}\times\big(\langle \mathsf{false}\rangle + \langle \mathsf{false}\rangle \times \frac{4}{5} + \langle \mathsf{false}\rangle \times \frac{2}{5}\big) \\
+ \\
\frac{1}{2}\times\big(\langle s = X \wedge n = 1\rangle + \langle n = 2\rangle \times \frac{4}{5} + \langle \mathsf{false}\rangle \times \frac{2}{5}\big)
\end{array}\right)
\end{array}$$

$\equiv$ embedded predicates and arithmetic

$$\begin{array}{l}
\frac{1}{\langle t{=}X\rangle} \times \big(\frac{2}{3} + \frac{1}{3} \times \big(\langle s = X \wedge n = 1\rangle + \langle n = 2\rangle \times \frac{2}{5}\big)\big) \\
\text{min} \\
\frac{1}{\langle t{\neq}X\rangle} \times \big(\frac{1}{2} \times \big(\langle s = X \wedge n = 1\rangle + \langle n = 2\rangle \times \frac{4}{5}\big)\big)
\end{array}$$

$\equiv$ $G_1 \wedge I_1 \ \Rightarrow \ n = 2$

$$\begin{array}{l}
\frac{1}{\langle t{=}X\rangle} \times \big(\frac{2}{3} + \frac{1}{3} \times \big(\langle s = X \wedge 2 = 1\rangle + \langle 2 = 2\rangle \times \frac{2}{5}\big)\big) \\
\text{min} \\
\frac{1}{\langle t{\neq}X\rangle} \times \big(\frac{1}{2} \times \big(\langle s = X \wedge 2 = 1\rangle + \langle 2 = 2\rangle \times \frac{4}{5}\big)\big)
\end{array}$$

$\equiv$ $\frac{1}{\langle t{=}X\rangle} \times \frac{4}{5} \quad \text{min} \quad \frac{1}{\langle t{\neq}X\rangle} \times \frac{2}{5}$. embedded predicates and arithmetic

D.2 Three Duelling Cowboys

We have the following information

$$E_2 \equiv \begin{array}{lcll} \frac{81}{140} & \times & \langle n = 3 \wedge t = X \rangle & + \\ \frac{83}{140} & \times & \langle n = 3 \wedge t = Y \rangle & + \\ \frac{27}{70} & \times & \langle n = 3 \wedge t = Z \rangle & + \\ & & \langle s = X \wedge n = 1 \rangle & \end{array} \tag{D.4}$$

The calculations are as follows (under the condition that $n = 3$:

$$\begin{array}{lcll} \frac{2}{3} & \times & [s \longleftarrow \mathsf{TwoCowboyZX}; n := 1] & E_2 \\ + & & & \\ (1 - \frac{2}{3}) & \times & [t := Y] & E_2 \end{array}$$

$\equiv$ Def. D.4

$$\begin{array}{lcll} \frac{2}{3} & \times & [(s \longleftarrow \mathsf{TwoCowboyZX}; n := 1)] & \\ & & \left(\begin{array}{lcll} \frac{81}{140} & \times & \langle n = 3 \wedge t = X \rangle & + \\ \frac{83}{140} & \times & \langle n = 3 \wedge t = Y \rangle & + \\ \frac{27}{70} & \times & \langle n = 3 \wedge t = Z \rangle & + \\ & & \langle s = X \wedge n = 1 \rangle & \end{array} \right) \\ + & \frac{1}{3} & \times & [t := Y] \\ & & \left(\begin{array}{lcll} \frac{81}{140} & \times & \langle n = 3 \wedge t = X \rangle & + \\ \frac{83}{140} & \times & \langle n = 3 \wedge t = Y \rangle & + \\ \frac{27}{70} & \times & \langle n = 3 \wedge t = Z \rangle & + \\ & & \langle s = X \wedge n = 1 \rangle & \end{array} \right) \end{array}$$

$\equiv$ simple and sequential substitution

$$\begin{array}{lcll} \frac{2}{3} & \times & [s \longleftarrow \mathsf{TwoCowboyZX}] & \\ & & \left(\begin{array}{lcll} \frac{81}{140} & \times & \langle 1 = 3 \wedge t = X \rangle & + \\ \frac{83}{140} & \times & \langle 1 = 3 \wedge t = Y \rangle & + \\ \frac{27}{70} & \times & \langle 1 = 3 \wedge t = Z \rangle & + \\ & & \langle s = X \wedge 1 = 1 \rangle & \end{array} \right) \\ + & \frac{1}{3} & \times & \left(\begin{array}{lcll} \frac{81}{140} & \times & \langle n = 3 \wedge Y = X \rangle & + \\ \frac{83}{140} & \times & \langle n = 3 \wedge Y = Y \rangle & + \\ \frac{27}{70} & \times & \langle n = 3 \wedge Y = Z \rangle & + \\ & & \langle s = X \wedge n = 1 \rangle & \end{array} \right) \end{array}$$

$\equiv$ logic, embedded predicate and arithmetic

$$\begin{array}{rcl} & \frac{2}{3} & \times \quad [s \longleftarrow \mathsf{TwoCowboyZX}]\,\langle s = X\rangle \\ + & \frac{1}{3} & \times \quad \big(\frac{83}{140} \times \langle n = 3\rangle + \langle s = X \wedge n = 1\rangle\big) \end{array}$$

$\equiv$ condition $n = 3$

$$\begin{array}{rcl} & \frac{2}{3} & \times \quad [s \longleftarrow \mathsf{TwoCowboyZX}]\,\langle s = X\rangle \\ + & \frac{1}{3} & \times \quad \big(\frac{83}{140} \times \langle 3 = 3\rangle + \langle s = X \wedge 3 = 1\rangle\big) \end{array}$$

$\equiv$ logic, embedded predicates and arithmetic

$$\frac{2}{3} \times [s \longleftarrow \mathsf{TwoCowboyZX}]\,\langle s = X\rangle + \frac{83}{420}\ .$$

Appendix E

List of publications

Below are list of publications related to this dissertation.

1. Thai Son Hoang, Zhendong Jin, Ken Robinson, Annabelle McIver, and Carroll Morgan. Probabilistic Invariants for Probabilistic Machines. In D. Bert, J. P. Bowen, S. King, and M. Waldén, editors, *ZB2003: Formal Specification and Development in Z and B, Proceedings of the 3rd International Conference of B and Z Users*, volume 2651 of *Lecture Notes in Computer Science*, Turku, Finland, June 2003. Springer Verlag.

2. Thai Son Hoang, Zhendong Jin, Ken Robinson, Annabelle McIver, and Carroll Morgan. Development via Refinement in Probabilistic B — Foundation and Case Study. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB2005: Formal Specification and Development in Z and B, Proceedings of the 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 355–373, Guildford, United Kingdom, April 2005. Springer Verlag.

3. Annabelle McIver, Carroll Morgan, and Thai Son Hoang. Probabilistic termination in B. In D. Bert, J. P. Bowen, S. King, and M. Waldén, editors, *ZB2003: Formal Specification and Development in Z and B, Proceedings of the 3rd International Conference of B and Z Users*, volume 2651 of *Lecture Notes in Computer Science*, Turku, Finland, June 2003. Springer Verlag.

4. Carroll Morgan, Thai Son Hoang, and Jean-Raymond Abrial. The Challenge of Probabilistic Event-B. In Helen Treharne, Steve King, Martin Henson, and

Steve Schneider, editors, *ZB2005: Formal Specification and Development in Z and B, Proceedings of the 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 162–171, Guildford, United Kingdom, April 2005. Springer Verlag.

5. Steve Schneider, Thai Son Hoang, Ken Robinson, and Helen Treharne. Tank Monitoring: A Case Study in pAMN. Technical report, Royal Holloway, the University of London, 2003. CSD-TR-03-17.

6. Steve Schneider, Thai Son Hoang, Ken Robinson, and Helen Treharne. Tank Monitoring: A pAMN Case Study. Electronic Notes in Theoretical Computer Science (ENTCS), April 2005.

7. Steve Schneider, Thai Son Hoang, Ken Robinson, and Helen Treharne. Tank Monitoring: A pAMN Case Study. *Formal Aspects of Computing*, 2005. Submitted for publication.