UNIVERSITY OF
Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

Performance-Oriented Refinement

Stefan Hallerstede

A thesis submitted for the degree of Doctor of Philosophy

Department of Electronics and Computer Science
University of Southampton
United Kingdom
July 2001

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

Doctor of Philosophy

Performance-Oriented Refinement

Stefan Hallerstede

We introduce the probabilistic action system formalism which combines refinement with performance. Performance is expressed by means of probability and expected costs. Probability is needed to express uncertainty present in physical environments. Expected costs express physical or abstract quantities that describe a system. They encode the performance objective. The behaviour of probabilistic action systems is described by traces of expected costs. Corresponding notions of refinement and simulation-based proof rules are introduced.

Formal notations like B [2] or action systems [8] support a notion of refinement. Refinement relates an abstract specification **A** to a more deterministic concrete specification **C**. Knowing **A** and **C** one proves **C** refines, or implements, specification **A**. In this study we consider specification **A** as given and concern ourselves with a way to find a good candidate for implementation **C**. To this end we classify all implementations of an abstract specification according to their performance. The performance of a specification **A** is a value val.**A** associated with some optimal behaviour it may exhibit. We distinguish performance from correctness. Concrete systems that do not meet the abstract specification are excluded. Only the remaining correct implementations **C** are considered with respect to their performance. A good implementation of a specification is identified by having some optimal behaviour in common with it. In other words, a good refinement corresponds to a reduction of non-optimal behaviour. This also means that the abstract specification sets a boundary val.**A** for the performance of any implementation. An implementation may perform worse than its specification but never better.

Probabilistic action systems are based on discrete-time Markov decision processes [98]. Numerical methods solving the optimisation problems posed by Markov decision processes are well-known, and used in a software tool that we have developed. The tool computes an optimal behaviour of a specification **A**, and the associated value val.**A**, thus assisting in the search for a good implementation **C**.

We present examples and case studies to demonstrate the use of probabilistic action systems.

Acknowledgement

It takes three to four years and a good supervisor to do a PhD. Michael Butler has been a very good one. I also enjoyed being part of the Declarative Systems and Software Engineering group at the University of Southampton headed by Peter Henderson. Ulrich Ultes-Nitsche and Eric Rogers have contributed to this work by reading my mini thesis and giving valuable comments. Finally, am grateful to Jeff Sanders and Ulrich Ultes-Nitsche for examining the final product of my PhD.

I began studying Computer Science at the University of Oldenburg in Germany. There I met Ernst-Rüdiger Olderog and Clemens Fischer who later supervised me during the writing of my diploma thesis. They are mainly responsible for my growing interest in formal methods during my studies in Oldenburg.

I would like to thank my family and various friends in Germany and the United Kingdom for giving joy to my private life. They distracted me from studying whenever it was necessary, and supported me in making important decisions. Helko Lehmann shared my flat while I was writing this thesis. He was in a similar situation [73]. He also was very good at distracting me from work but also at giving useful advice.

Contents

1 Introduction **6**
1.1 State-Based Approaches 7
1.2 Event-Based Approaches 8
1.3 Example: Bookshop Inventory 10
1.4 Overview 13

2 Foundations **14**
2.1 Sets 14
2.2 Mathematical Notation 16
2.3 Probability 17
2.4 Action Systems 17
2.4.1 Syntax 17
2.4.2 Semantics 18
2.4.3 Failure Refinement 19

3 Program Semantics **21**
3.1 State Functions 22
3.2 State Relations 22
3.3 Probabilistic State Functions 24
3.4 Probabilistic State Relations 29
3.5 Extended Probabilistic State Relations 33
3.6 Algebraic Laws 36
3.7 Remarks 42

4 Probabilistic Action Systems **44**
4.1 Behaviour of Probabilistic Action Systems 45
4.2 Syntactic Representation 46
4.3 Example: Polling System 47
4.4 Cost Refinement 49
4.4.1 Simulation 51
4.4.2 Equivalence 56
4.5 Example: Polling System 59
4.6 Remarks 66

5 Optimal Systems **68**
5.1 Markov Decision Processes . . . 68
5.2 Optimisation Criteria . . . 70
5.3 Average Cost Optimality . . . 72
5.4 Example: Polling System . . . 74
5.5 Remarks . . . 77

6 DYNAS **80**
6.1 Compiler . . . 81
6.2 Expander . . . 85
6.2.1 Data Structures . . . 85
6.2.2 Game Semantics . . . 88
6.3 Solver . . . 94
6.3.1 Policy Iteration . . . 94
6.3.2 Value Iteration . . . 97
6.4 Printer . . . 99

7 Case Study: Lift System **102**
7.1 State and Operation of the Lift Machine . . . 102
7.2 From Machines to Systems . . . 105
7.2.1 Interleaving . . . 106
7.2.2 Random Actions . . . 106
7.2.3 Dependent Actions . . . 108
7.2.4 Independent Actions . . . 109
7.2.5 The Lift System . . . 111
7.3 Time Scale Transformation . . . 112
7.4 More Actions . . . 112
7.5 Fewer States . . . 114
7.5.1 State Aggregation . . . 114
7.5.2 The Reduced Lift System . . . 118
7.6 Evaluation and Discussion . . . 119

Conclusion **121**

A Proofs **124**

B Mathematical Notation **144**

C ASCII-Representation **146**

Index **149**

Bibliography **151**

Chapter 1

Introduction

In recent years there has been growing interest in combining formal methods with performance analysis. The resulting developments gave rise to stochastic variants of established event-based and state-based formalisms. We distinguish between event-based and state-based formalisms by way of their behavioural semantics. The B-formalism [2] can be considered event-based [3]. It lacks a means of performance analysis. This thesis describes our effort to supplement B with a suitable notion of performance. From the outset we have only considered extensions that would support automatic calculation of performance measures. Realistically sized systems usually consist of thousands of states leading to thousands of equations to be solved. We consider it infeasible to do this by hand. Our extension of B, probabilistic action systems, is not event-based anymore (see chapter 4). Neither events nor states are observable. The behaviour is described in terms of expected costs which capture certain features that are relevant for performance. Despite this difference, refinement of probabilistic action systems is reminiscent of B refinement. This makes it easier to learn for someone familiar with B refinement already. In practice, this will also mean that experience in either is useful in the other.

In our current study we measure performance in terms of long-run expected average-cost. Probabilistic action systems do not make assumptions about any particular performance measure. It ought to be possible to choose a measure suitable for the system being investigated. Syntactically probabilistic action systems are close to B [2] and probabilistic predicate transformers [88]. The program constructs used also lean on [9]. We decided on B as a foundation of probabilistic action systems because of its widespread use. The inclusion of features for performance analysis was made as non-intrusive as possible. We believe the notation used for performance analysis should be close to the notation used in the development process. Then specifications used in B-refinement, or parts thereof, could be used in performance analysis and results from the analysis can easily be transferred back.

Sections 1.1 and 1.2 review state-based and event-based approaches to performance analysis. We shall briefly discuss notions of performance, refinement and simulation used in the mentioned formalisms. In section 1.3 we demonstrate probabilistic action systems by way of an inventory problem, and compare the modelling approach to those presented in sections 1.1 and 1.2. Finally, section 1.4 gives an overview of the remaining chapters.

1.1 State-Based Approaches

The models underlying the formalisms discussed in this and the next section are called Markov processes [61, 70] and Markov decision processes [62, 115]. The two models themselves are used to model state-based stochastic systems [16, 98, 110]. However, they are impractical to use in the performance analysis of complex systems because no structuring mechanisms are available [45]. Queueing systems are the traditional structured formalism used in performance analysis [84]. They have also been applied to computer systems performance modelling [24, 43, 69, 92, 106]. Usual performance measures derived from queueing systems include: system throughput, average numbers of waiting customers at stations in a queueing network, and waiting times [84].

Closely related to performance analysis is the subject of "performability". Performability integrates performance and reliability modelling. Its purpose is the performance analysis of degradable systems. To a limited degree performability analysis is possible with our formalism because of its semantical model. There is no explicit support for performability analysis though. The articles [44, 45] give an overview of existing formalisms and tools for performability analysis. One such tool is presented in [106] together with an introduction to some formalisms. These include queueing systems and stochastic Petri nets which are widely used in performance analysis, and some formalisms specific to reliability analysis.

Stochastic Petri nets [79, 85] have been applied to performance modelling of computer architectures. Their origin are classical Petri nets [101, 113] which are described by a collection of places, transitions and markings. Exponentially distributed firing delays are used to model uncertain behaviour. Consequently, stochastic Petri nets model real-time and probability. There is no notion of nondeterminism though. The operational behaviour of (generalised) stochastic Petri nets is characterised by the interaction of immediate transitions and exponentially delayed transitions. If two exponentially delayed transitions t_1 and t_2 in a stochastic Petri net compete for a token, the conflict is resolved probabilistically. Let μ_1 and μ_2 be the transition rates of t_1 and t_2, i.e. the mean time it takes for t_i to fire is $1/\mu_i$. Then transition t_i fires with probability $p_i = \mu_i/(\mu_1 + \mu_2)$. The use of the transition probabilities p_i corresponds to a shift to discrete time [84]. If an imme-

diate transition is enabled in a marking, that marking "vanishes", i.e. in the semantical model the marking is not visible. If an immediate transition conflicts with an exponentially delayed one, the immediate transition has priority. And if two immediate transitions conflict the conflict is resolved by explicitly specified priorities.

The generalised model [79] added immediate transitions to the original model [85]. In [75] the generalised stochastic Petri nets of [79] are extended with a type of transition having deterministically distributed firing delays. The extended stochastic Petri nets are mainly used for computer architectures modelling [75]. The book [75] contains another overview over performance modelling techniques, and presents a software tool [74] that computes performance measures. The performance measures that can be derived from stochastic Petri nets are expectations of functions of markings, and probabilities of predicates over markings [75]. In [75] the lack of nondeterminism in these formalism is partly remedied by the use of parameterised specifications, e.g. experiments in [75]. But there is no means to reason about these parameters from within the formalism. Chapters on stochastic Petri nets are also contained in [43, 69]. In [69] some consideration is given to discrete-time stochastic Petri nets as well. None of these state-based formalisms is accompanied by a notion of refinement.

The action system formalism [8] has been extended in [111] with probabilistic features for reliability analysis. It is based on the probabilistic extension [88] of the guarded command language [29]. The probabilistic guarded command language contains notions of nondeterminism and probabilistic choice but is not compatible with the general performance measures supported by Markov decision processes and used in our approach. In [116] this has been partly rectified by using parameterised refinement similar to [75]. By insisting on a close correspondence to standard probability theory our approach is similar to [49]. However, our model is closely based on Markov decision processes so that tool support can be easily achieved.

1.2 Event-Based Approaches

The event-based formalisms for performance analysis are usually based on classical process algebras like CCS [83], CSP [59] or LOTOS [19]. They are generally called stochastic process algebras, e.g. EMPA [14, 13, 15], MPA [20], TIPP [53, 54], or PEPA [57, 58].

Stochastic process algebras are deterministic in the sense that for all choices there are (stochastic) instructions how to resolve them. Similar to stochastic Petri nets, stochastic process algebras use exponentially distributed delays between events. Their behaviour is usually described by labelled transition systems or traces of actions. Let μ denote a rate and a an action. An activity α is defined by a tuple (a, μ). A choice between

two activities $\alpha_1 = (a_1, \mu_1)$ and $\alpha_2 = (a_2, \mu_2)$ is resolved similarly to conflict resolution in stochastic Petri nets: action a_i occurs with probability $\mu_i/(\mu_1 + \mu_2)$. Stochastic process algebras have the usual combinators, like synchronisation or hiding. The definition of synchronised composition varies between the different algebras. Hidden actions are called internal, and are denoted by the special symbol τ. Internal actions themselves are not observable, only their effect is. As in classical process algebras notions of bisimulation and equivalence between process terms exist. These form the basis of methods to reduce the size of the semantical model of process terms for numerical analysis [51, 58].

The stochastic process algebras TIPP and EMPA also have notions of nondeterminism. In fact, the stochastic process algebra EMPA has language kernels that correspond to classical process algebra like CCS, stochastic process algebras like MPA, and probabilistic process algebras like probabilistic CSP [108]. However, in EMPA performance analysis is only possible for specifications that do not contain nondeterminism. In TIPP the process term being analysed must be bisimilar to a deterministic process term, effectively saying the original process term does not contain nondeterministic choices.

The semantics of process algebras can be described in terms of Petri nets [94]. The same holds for stochastic process algebras and stochastic Petri nets [11, 13, 102, 103]. This relationship is useful to compare the two modelling approaches, and to transfer concepts between them. This also means that performance measures used in one approach are, in principle, also available in the other one. Typical measures used with stochastic process algebras are probabilities of process states, throughput, and means based on variables present in parametric processes [51]. In [23] and [10, 12] the stochastic process algebras PEPA and EMPA have been equipped with means to specify more general performance measures based on rewards [62]. The approach of [23] is to use temporal logic-like formulas to refer to states of a system (that are determined by sequences of actions), and assign rewards to them. In EMPA$_r$ [12] activities are triples (a, μ, r) where a and μ are as above and $r \in \mathbb{R}$ is a reward associated with action a. The article [12] gives examples on how to express standard performance measures as mentioned above in EMPA$_r$. This is extended in [10] to include also rewards associated with states. Tool support for any of these methods is generally considered to be essential for the method to be useful in practice [44]. Software tools are available to compute performance measures specified in the process algebras PEPA: PEPA Workbench [37], TIPP: TIPPtool [51], and EMPA$_r$: TwoTowers [10]. The tool TwoTowers also supports some model checking of functional aspects specified in EMPA$_r$ process terms.

system $BOOKSHOP$
constants
$\quad BOOKS;\ \ CAPACITY;\ \ CUSTOMERS;$
constraints
$\quad BOOKS = 3 \wedge CAPACITY > 0 \wedge CUSTOMERS > 0;$
sets
$\quad BOOK \ = \ 1\,..\,BOOKS;$
$\quad PRICE \ = \ \langle 5.0, 70.0, 20.0 \rangle;$
$\quad DEMAND \ = \ \langle 0.2, 0.13, 0.34, 0.33 \rangle;$
$\quad CUSTOMER \ = \ 1\,..\,CUSTOMERS;$
variables
$\quad stock \ : \ BOOK \to 0\,..\,CAPACITY;$
programs
$\quad replenish \ =$
$\qquad \bigsqcup RR : BOOK \to 0\,..\,CAPACITY \bullet$
$\qquad\quad |\ \forall\, bb : BOOK \bullet stock.bb + RR.bb \leq CAPACITY\ |;$
$\qquad\quad stock := (\lambda\, bb : BOOK \bullet stock.bb + RR.bb);$
$\quad sell \ =$
$\qquad \bigoplus DD : CUSTOMER \to BOOK \cup \{BOOKS + 1\}$
$\qquad |\ (\prod cc : CUSTOMER \bullet DEMAND.(DD.cc)) \bullet$
$\qquad\quad |\ \sum bb : BOOK \bullet (\mathsf{card}.(DD^{\sim}[\{bb\}]) \dot{-} stock.bb) * PRICE.bb\ |;$
$\qquad\quad stock := (\lambda\, bb : BOOK \bullet stock.bb \dot{-} \mathsf{card}.(DD^{\sim}[\{bb\}]));$
initialisation
$\quad stock := BOOK \times \{0\};$
actions
$\quad trade \ =$
$\qquad replenish; sell;\ |\ \sum bb : BOOK \bullet stock.bb * PRICE.bb\ |;$
end

Figure 1.1: A small book shop

1.3 Example: Bookshop Inventory

We introduce probabilistic action systems with an example of an inventory control problem based on [98, 110]. This section mainly serves two purposes. First, it describes the specification style we use and, second, demonstrates how modelling problems from standard literature may be approached using probabilistic action systems.

A book shop sells three different $BOOKS$. It may hold up to $CAPACITY$ copies of a book in store. Every day a number of $CUSTOMERS$ visits the shop some of which buy a copy of a book. See figure 1.1. The price of book bb is given by $PRICE.bb$. It is known that 20% of the customers buy book 1, 13% buy book 2, 34% buy book 3, and 33% of them leave the shop without

buying anything. This is expressed by the sequence $DEMAND$. Note that the last element of the sequence $DEMAND.(BOOKS + 1)$ represents the fraction of customers not buying a book. No customer buys more than one book a day. The stock of the shop is represented by variable *stock*: there are $stock.bb$ copies of book bb in stock. The book shop gets a delivery of books every morning which are subsequently sold on to customers. We regard a request for a book that cannot be served as a loss, i.e. a cost the shop incurs. However, holding many books in stock also incurs costs. We assume that both types of costs are given by the resale prices of the corresponding books. Performance objectives are associated with system $BOOKSHOP$ by associating costs with its states. The objective is to keep the operating costs of the shop as low as possible, that is, cheaper states are to be preferred during operation.

The operation of system $BOOKSHOP$ proceeds by first executing its initialisation, and afterwards repeatedly its action *trade*. First the program $stock := BOOK \times \{0\}$ is executed, i.e. initially the stock is empty. Then the book shop buys, stores, and sells books. The only action *trade* of $BOOKSHOP$ is split into two programs *replenish* and *sell* which are followed by a cost statement:

$$| \sum bb : BOOK \bullet stock.bb * PRICE.bb \mid . \tag{1.1}$$

When this statement is encountered in an execution the specified cost is incurred. The above cost statement (1.1) values the stock of the book shop that has not been sold during the day. The implied performance objective is to minimise the value of stock that is kept unnecessarily. A conflicting objective is specified in program *sell*, incurring costs if demand is not met. Improving on one objective worsens the other one.

Program *replenish* models the delivery of books to the book shop in the morning. The assumption behind this program is that the books have been ordered the evening before and they arrive in the morning. The shop manager can only order as many books as can be stored in the shop, present stock included. Orders are represented by function $RR : BOOK \rightarrow CAPACITY$ which details the amount of copies of each book to order. The nondeterministic choice $\bigsqcup$ over possible orders RR describes the decision on an order RR to be made by the shop manager. An implementation of system $BOOKSHOP$ would replace this choice by a particular choice, for instance, one that minimises the running costs of the shop. That implementation would correspond to the shop manager adopting a policy of running the book shop.

Program *sell* models the behaviour of customers. Each customer cc buys a copy of a book $DD.cc \in 1 \mathinner{..} BOOKS$, or none. The latter case is modelled by the “dummy” book identifier $BOOKS + 1$. Function DD models the demand of a number of $CUSTOMERS$ over one day. The product $\prod_{cc} DEMAND.(DD.cc)$ describes the joint probability that each customer

goal	*cost*	*CUSTOMERS*
$\langle 0,0,1 \rangle$	31.22	2
$\langle 0,0,1 \rangle$	42.18	3
$\langle 1,0,1 \rangle$	54.28	4
$\langle 1,1,2 \rangle$	65.46	5
$\langle 1,1,2 \rangle$	66.56	6
$\langle 1,1,2 \rangle$	70.44	7

Table 1.1: Optimal stock-keeping

cc buys book $DD.cc$, or no book if $DD.cc = BOOKS + 1$. Using this model the following property holds:

$$\sum_{DD}(\prod_{cc} DEMAND.(DD.cc)) \;=\; 1 \;.$$

The probabilistic choice $\bigoplus$ models the stochastic behaviour of all customers over one day, assigning probability $\prod_{cc} DEMAND.(DD.cc)$ to demand DD. As mentioned earlier an attempt of customer cc to buy a book $bb = DD.cc$ that is not in stock corresponds to a loss of $PRICE.bb$ in revenue. For an entire demand DD this equals:

$$|\; \sum bb : BOOK \bullet (\mathsf{card}.(DD^{\sim}[\{bb\}]) \dot{-} stock.bb) * PRICE.bb \;|\;. \qquad (1.2)$$

The symbol $\dot{-}$ denotes subtraction bounded below by zero:

$$xx \dot{-} yy \;\;\widehat{=}\;\; xx - \mathsf{min}.\{xx, yy\}\;.$$

Hence, the expression $\mathsf{card}.(DD^{\sim}[\{bb\}]) \dot{-} stock.bb$ describes the number of missing copies of book bb to meet the total demand $DD^{\sim}[\{bb\}]$ of book bb. To minimise (1.2) the shop manager would keep as many copies in stock as possible. This is the opposite of what is required in (1.1), i.e. trying to keep the storage costs low. Taking both objectives into account the manager seeks a trade-off between them.

Using the software tool described in chapter 6, we can calculate the optimal stock to be kept with respect to the long-run average-cost incurred by the system per day. Letting $CAPACITY = 3$, table 1.1 presents the stock an optimally operating book shop would keep. For instance, if one expects 6 customers per day: keep one copy of book 1, one copy of book 2, and two copies of book 3. The average cost incurred is approximately 66.56 per day. An optimal implementation of $BOOKSHOP$ will always issues orders such that each morning, after delivery, $stock = goal$.

If refinement was only used in this way it would not be powerful enough. It becomes more valuable if it supports a change in the representation of the state space. In this thesis we use this aspect of refinement for state

aggregation, a technique also used in stochastic process algebras (see section 1.2). In this context refinement is employed to reduce the size of the state space of probabilistic action systems to facilitate automatic analysis. Occasionally the model of time underlying probabilistic action systems is referred to as real-time [121]. However, we prefer to say it is discrete-time in correspondence with the majority of formalisms in sections 1.1 and 1.2, and the traditional terms used in stochastic dynamic programming [16, 98].

1.4 Overview

Action systems and trace refinement are briefly introduced in chapter 2. They are the foundation of this work. In the same chapter we also introduce notations and conventions used throughout the thesis.

Chapter 3 describes the program notation and semantics we use. A set of program models is introduced in form of a small hierarchy. We think this presentation of the semantics of programs makes it easier to understand the different modelling aspects involved.

Based on the expectation-based program model of section 3.5 probabilistic action systems are described in chapter 4. In the same chapter cost refinement of probabilistic action systems is presented. Cost refinement has been developed from trace refinement of action systems described in chapter 2. Probabilistic action systems and cost refinement are explained by way of an extended example that is continued in chapter 5.

In chapter 5 we explain the principle of optimality as applied in the theory of stochastic dynamic programming [28]. Average cost optimality of probabilistic action systems and of Markov decision processes [28] are identified. This opens the way for automatic performance analysis of probabilistic action systems (see chapter 6). We also relate cost refinement with average cost optimality of probabilistic action systems. The presented material is demonstrated with the continuation of the example of chapter 4.

Chapter 6 describes the implementation of a software tool that computes an optimal implementation of probabilistic action system. The software tool utilises results from chapter 5 substituting probabilistic action systems by Markov decision processes in computations.

In chapter 7 a case study is presented to demonstrate how probabilistic action systems may be used in practice. None of the refinements proved in this chapter is actually aimed at implementation. Refinement is only used for state aggregation. Otherwise analysis with the software tool would be impossible. We also introduce a transformation for probabilistic action systems that preserves average cost optimality. This transformation is necessary to enable the use of the software tool. The transformation is not a refinement. Yet, we prove that an optimal implementation of the transformed system is also an optimal implementation of the original one.

Chapter 2

Foundations

This chapter is not intended as an introduction to the mentioned subjects. Its purpose is to familiarise the rare reader of this document with the formal notions and concepts used. We use a notation based on type theory to define program semantics similar to [9]. Specifications are written in a B-like notation [2] which is based on set theory. Section 2.1 bridges the two approaches. So we really use a single notation throughout the text. Section 2.2 introduces some more notation required later, and section 2.3 gives some references to background material on probability theory. In section 2.4 we describe the syntax and semantics of action systems. We also briefly discuss the associated notion of refinement.

2.1 Sets

We assume a universe $\mathcal{U}$ of nonempty sets. On objects in this universe the usual set operations set intersection $\cap$, set union $\cup$, set difference $\setminus$, and the relationships set member $\in$, and set equality $=$, are defined. Set cardinality is denoted by card.

Truth Values

The set of truth values $\{\mathsf{true}, \mathsf{false}\}$ is denoted by $\mathbb{B}$. The usual logical operators $\vee$, $\wedge$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$, are defined on $\mathbb{B}$.

Natural Numbers

The set of natural numbers is denoted by $\mathbb{N}$:

$$\mathbb{N} \;=\; \{0, 1, 2, \ldots\} \;.$$

The set of natural numbers without 0 is denoted by $\mathbb{N}_1$, i.e. $\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$. The interval of the natural numbers from m to n is defined by

$$m \,..\, n \;\widehat{=}\; \{k \mid m \leq k \leq n\} \;.$$

Custom Values

Custom sets S of finitely many values $V_1,\ V_2,\ \ldots,\ V_k$ are defined by enumeration of the contained constants:

$$S \ = \ V_1 \mid V_2 \mid \ldots \mid V_k \ .$$

This corresponds to a matching set in the universe $\mathcal{U}$.

Real Numbers

The set of real numbers is denoted by $\mathbb{R}$. The open real interval from x to y, where $x \leq y$, is denoted by (x, y). We allow the values $-\infty$ and $+\infty$ for x and y, so that $\mathbb{R} = (-\infty, +\infty)$. The set of positive real numbers is defined by $\mathbb{R}_{\geq 0} = \{0\} \cup (0, +\infty)$.

Types

Types are expressions that describe sets. A type is described by a nullary type operator C, or one of the type operators below. A nullary type operator C is a subset of $\mathbb{B}$, $\mathbb{N}$, $\mathbb{R}$, or a set S of custom values. The other type operators denote function types $\Delta_1 \rightarrow \Delta_2$, partial function types $\Delta_1 \nrightarrow \Delta_2$, relation types $\Delta_1 \leftrightarrow \Delta_2$, product types $\Delta_1 \times \Delta_2$, and power types $\mathbb{P}\,\Delta_1$. Generally, types are defined by the following grammar:

$$\Delta \ = \ C \mid \Delta_1 \rightarrow \Delta_2 \mid \Delta_1 \nrightarrow \Delta_2 \mid \Delta_1 \leftrightarrow \Delta_2 \mid \Delta_1 \times \Delta_2 \mid \mathbb{P}\,\Delta_1 \ .$$

The type $\Delta_1 \times \Delta_2$ denotes a set of pairs. Pairs are denoted by (x, y), and alternatively $x \mapsto y$. The latter is the preferred notation in connection with function types. Members of types $\Delta_1 \rightarrow \Delta_2$, $\Delta_1 \nrightarrow \Delta_2$ and $\Delta_1 \leftrightarrow \Delta_2$ correspond to sets of pairs. If f is of function or relation type, then $\mathsf{dom}.f$ denotes its *domain* and $\mathsf{ran}.f$ its *range*. Function application is denoted $f.x$, where $x \in \mathsf{dom}.f$. The type $\mathbb{P}\,\Delta_1$ denotes the power set of Δ_1. The set operators of the universe are used with types. By $\mathbb{P}_1\,\Delta_1$ we denote the set $\mathbb{P}\,\Delta_1 \setminus \{\varnothing\}$ of non-empty subsets of Δ_1.

Notation

A function $q : \Delta \rightarrow \mathbb{B}$ is called a predicate. Predicates and power types are isomorphic,

$$\mathbb{P}\,\Delta \ \simeq \ \Delta \rightarrow \mathbb{B},$$

if we let $x \in q \Leftrightarrow q.x$. This gives the usual correspondence between the two:

$$(q_1 \cup q_2).x \ \Leftrightarrow \ q_1.x \vee q_2.x \ ,$$
$$(q_1 \cap q_2).x \ \Leftrightarrow \ q_1.x \wedge q_2.x \ ,$$

$$\begin{aligned}(\Delta \setminus q_1).x &\Leftrightarrow \neg\, q_1.x \ ,\\ (q_1 \subseteq q_2).x &\Leftrightarrow q_1.x \Rightarrow q_2.x \ ,\\ (q_1 = q_2).x &\Leftrightarrow q_1.x \Leftrightarrow q_2.x \ .\end{aligned}$$

Universal and existential quantification are defined as usual:

$$\begin{aligned}(\forall\, x : \Delta \bullet q) &\Leftrightarrow q.x = \mathsf{true} \text{ for all } x \in \Delta \ ,\\ (\exists\, x : \Delta \bullet q) &\Leftrightarrow q.x = \mathsf{true} \text{ for some } x \in \Delta \ .\end{aligned}$$

We often use relations of type $\Delta_1 \leftrightarrow \Delta_2$ like functions based on the isomorphism

$$\Delta_1 \leftrightarrow \Delta_2 \ \simeq \ \Delta_1 \rightarrow \mathbb{P}\,\Delta_2,$$

where $(x, y) \in r \Leftrightarrow y \in r.x$. Observe that using the isomorphism between power types and predicates this extends to $(x, y) \in r \Leftrightarrow r.x.y$. This notation proves very convenient in definitions on the semantical level which is the reason why it is introduced. In specifications we treat relations (and functions) as sets of pairs though. With few exceptions the two notations are not mixed.

2.2 Mathematical Notation

In addition to the types introduced in section 2.1 we use finite sequence types

$$\begin{aligned}\mathsf{seq}[n]\Delta &\ \widehat{=}\ \{s \in \mathbb{N}_1 \nrightarrow \Delta \mid \mathsf{dom}.s = 1 \,..\, n\} \ ,\\ \mathsf{seq}\,\Delta &\ \widehat{=}\ \textstyle\bigcup_{n \in \mathbb{N}} \mathsf{seq}[n]\Delta \ ,\end{aligned}$$

and infinite sequence types

$$\mathsf{seq}_\infty\,\Delta \ \widehat{=}\ \mathbb{N}_1 \rightarrow \Delta \ .$$

Refer to appendix B for the mathematical notation used in conjunction with sequences, relations, etc. The notation is close to B [2] and Z [112].

We also use the same notation for substitution as B. The expression

$$[x := e]\, q$$

denotes the expression q with all free occurrences of x replaced by e. Simultaneous substitution is denoted by

$$[x_1 := e_1, x_2 := e_2]\, q \ .$$

2.3 Probability

Familiarity with fundamental concepts of linear algebra [76], analysis [56], and probability theory is assumed. We use the notation $\mathbf{P}(\ldots)$ to denote probability distributions, and the notation $\mathbf{E}(\ldots)$ to denote expectations. All material presented in this work concerns discrete probability theory only. For probability theory see [80, 84, 115, 118]. Stochastic dynamic programming and Markov decision processes are treated in [16, 28, 98, 110, 115].

2.4 Action Systems

A variety of specification formalisms have been proposed as modelling languages for distributed reactive systems [1, 3, 21, 22, 67]. We refer to the state-based formalisms [3, 21] generally as action systems although they differ somewhat from the formalism originally introduced in [5]. In the same manner we refer to the formalism introduced in this chapter as action systems. It is similar to the B-derivative presented in [3]. It is based on relational semantics similar to [33], though, instead of predicate transformer semantics used in [3, 21].

Action systems are the conceptual foundation of probabilistic action systems which are presented in chapter 4. Yet the behavioural semantics of the two differs significantly. The semantics of action systems is described in terms of the failures model of CSP [59, 104, 105] or in form of the state-trace model [8], whereas the semantics of probabilistic action systems is defined in terms of cost traces. The case study in chapter 7 relates the two formalisms in a practical example. Adopting B-terminology, we often refer to action systems using the term *machine*. The term *system* is reserved for reference to probabilistic action systems, and for informal discussions involving, for instance, queueing systems.

2.4.1 Syntax

The specification of an action system is partitioned into different sections describing different aspects of it. Figure 2.1 shows the syntactic structure of an action system with name *TIMER*. It models a timer that counts to T, and then stops. The constants section declares natural number constants the value of which is restricted in the constraints section. The sets section declares sets that are used to constrain the possible values of variables. Variables are declared in the variables section. Their cartesian product makes up the state space of an action system. Initial values of the variables are specified in the initialisation section. The initialisation program must not assume that there is a state before its execution. The last section actions specifies the operational part of the action system. It consists of a number of actions each of which may have a list of comma-separated parameters.

$\mathsf{machine}\ \mathit{TIMER}$
$\mathsf{constants}\ T;$
$\mathsf{constraints}\ T \geq 1;$
sets
$\quad \mathit{TIME} \ = \ 1 \mathrel{..} T;$
$\mathsf{variables}$
$\quad \mathit{time} \ : \ \mathit{TIME} \cup \{T+1\};$
$\mathsf{initialisation}$
$\quad \mathit{time} := 1;$
$\mathsf{actions}$
$\quad \mathit{tick}(t : \mathit{TIME}) \ =$
$\qquad | \ \mathit{time} = t \ |; \mathit{time} := \mathit{time} + 1;$
$\quad \mathit{tock}(t : \mathit{TIME}) \ =$
$\qquad | \ \mathit{time} = t \ |; (\mathsf{skip} \sqcup \mathit{time} := \mathit{time} + 1);$
end

Figure 2.1: Syntax of an action system

The actions describe possible state changes of the action system. For the program notation used in the initialisation and actions sections see sections 3.1 and 3.2 which can be read independently of the rest of chapter 3.

2.4.2 Semantics

The behaviour of an action system is described by a set of failures. A *failure* (t, X) consists of a finite sequence t of events, and a set X of *refusals*. We refer to action names with fully specified parameter values, e.g. $\mathit{tick}(1)$, as events. The set of events an action system $\mathbf{M}$ may engage in is called its *alphabet*, denoted $\mathsf{alpha}.\mathbf{M}$. Sequence t is usually called a *trace*. If (t, X) is a failure of some action system $\mathbf{M}$, then $\mathbf{M}$ may first engage in trace t, and afterwards refuse to engage in any event contained in X. When an action system engages in an event, say, $\mathit{tick}(1)$ the corresponding program, $| \ \mathit{time} = 1 \ |; \mathit{time} := \mathit{time} + 1$, is executed on its state space. Examples of failures of machine *TIMER* follow. The tuples

$$(\langle \mathit{tock}(1) \rangle, \{\mathit{tick}(1)\}) \text{ and } (\langle \mathit{tock}(1) \rangle, \{\mathit{tick}(2)\})$$

are failures of machine *TIMER*, assuming that T is greater than 1. Whereas the tuple

$$(\langle \mathit{tock}(1) \rangle, \{\mathit{tick}(1), \mathit{tick}(2)\})$$

is not a failure of *TIMER*, because after having engaged in event $\mathit{tock}(1)$ machine *TIMER* cannot refuse both of the events $\mathit{tick}(1)$ and $\mathit{tick}(2)$. This

kind of behaviour is referred to as *internal* nondeterminism. It is internally determined by machine *TIMER* which events may occur next. The second kind of nondeterminism present in action systems is called *external* nondeterminism. It denotes choices that are entirely determined by the environment of an action system. The testing preorders of [50], and the explicit representation of internal choice in [83], are intuitive alternative views to the failures model we use. The definition of failures below follows the approach in [67, 39] which is based on relational programs. In [86, 119] a similar approach is taken using predicate transformers instead. The model based on predicate transformers contains failures and divergences [59]. The model presented here does not deal with divergence because the relational programs used cannot cause divergent behaviour.

Let $t \in \mathsf{seq}(\mathsf{alpha}.\mathbf{M})$ and $a \in \mathsf{alpha}.\mathbf{M}$. We denote by $\mathbf{M}.a$ the action corresponding to event a with its parameters instantiated as specified by a. The set of states that are reachable by executing a trace of actions t of machine $\mathbf{M}$ is defined by:

$$\begin{aligned}\mathsf{trace}.\mathbf{M}.\langle\,\rangle &\mathrel{\widehat{=}} \mathsf{ran}.(\mathbf{M}.\mathsf{initialisation}) \\ \mathsf{trace}.\mathbf{M}.t^\frown\langle a\rangle &\mathrel{\widehat{=}} (\mathbf{M}.a)[\mathsf{trace}.\mathbf{M}.t] \ .\end{aligned}$$

We say t is a trace of machine $\mathbf{M}$ if $\mathsf{trace}.\mathbf{M}.t \neq \varnothing$. Refusals are defined relative to the state of a machine $\mathbf{M}$ as outlined above. Let $X \subseteq \mathsf{alpha}.\mathbf{M}$. The set of events X is a refusal in state τ if all of the programs $\mathbf{M}.a$ corresponding to the events a in X block execution in state τ. The definition of $\mathsf{refusal}.\mathbf{M}$ follows.

$$\mathsf{refusal}.\mathbf{M}.\tau.X \ = \ \forall\, a : X \bullet \tau \notin \mathsf{dom}.(\mathbf{M}.a) \ .$$

Finally we define the failures of machine $\mathbf{M}$. The tuple (t, X) is a failure of $\mathbf{M}$ if all events in X can be refused in a state that is reachable after engaging in trace t. Formally,

$$\mathsf{failure}.\mathbf{M}.(t, X) \ \widehat{=} \ \exists\, \tau : \Gamma \bullet \mathsf{trace}.\mathbf{M}.t.\tau \wedge \mathsf{refusal}.\mathbf{M}.\tau.X \ .$$

We also refer to $\mathsf{failure}.\mathbf{M}$ as the behaviour of machine $\mathbf{M}$.

2.4.3 Failure Refinement

A machine **MB** is said to refine, or implement, another machine **MA** if the behaviour of **MB** is subsumed by the behaviour of **MA**. In the context of failure refinement this means, whenever **MB** can engage in a trace t, so can **MA**; and whenever **MB** can refuse a set of events X, then **MA** can also refuse X. We denote by $\mathbf{MA} \sqsubseteq \mathbf{MB}$ that machine **MB** refines **MA**, and define

$$\mathbf{MA} \sqsubseteq \mathbf{MB} \ \widehat{=} \ \mathsf{failure}.\mathbf{MB} \subseteq \mathsf{failure}.\mathbf{MA} \ .$$

If $\mathbf{MA} \sqsubseteq \mathbf{MB}$ holds we refer to $\mathbf{MA}$ as the *abstract* machine and $\mathbf{MB}$ as the *concrete* machine.

Usually refinements are not proven using the definition itself. Preferably simulation techniques are employed. A simulation imitates the step by step behaviour of a machine. We present a simulation in propostion 2.1 below. A soundness proof can be found in [39, 46]. In proposition 2.1 let $\mathbf{M}.\mathsf{initialisation}$ denote the set of initial states of machine $\mathbf{M}$. This is not a problem because we are assuming that the initialisation of a machine does not refer to variables that have not been initialised.

Proposition 2.1 Let $\mathbf{MA}$ and $\mathbf{MB}$ be machines, and let *sim* be a relational program with the state space of $\mathbf{MA}$ as its domain and the state space of $\mathbf{MB}$ as its range. If (for all actions *act* of the two machines) the conditions

$$\mathbf{MB}.\mathsf{initialisation} \subseteq sim[\mathbf{MA}.\mathsf{initialisation}] \qquad \text{(MS1)}$$

$$sim[\mathsf{dom}.(\mathbf{MA}.act)] \subseteq \mathsf{dom}.(\mathbf{MB}.act) \qquad \text{(MS2)}$$

$$sim; \mathbf{MB}.act \subseteq \mathbf{MA}.act; sim \qquad \text{(MS3)}$$

hold, then $\mathbf{MA} \sqsubseteq \mathbf{MB}$.

Condition (MS1) requires that the abstract machine is able to simulate the initialisation of the concrete machine. Condition (MS2) requires that whenever an abstract action can be executed, then also the concrete action can be executed in a corresponding state. The last condition (MS3) requires that abstract actions can simulate concrete actions.

As a proof rule proposition 2.1 is sound but not complete. Rule 2.1 is usually called forward simulation. Completeness can be achieved by introducing a second rule called backward simulation [47, 60, 67, 119].

Chapter 3

Program Semantics

We introduce several semantic models for different classes of programs. To do this turns out to be useful in two ways. Firstly, each model describes a certain class of programs. If we need a program belonging to that class we simply refer to the corresponding model. Secondly, the models are naturally arranged in a hierarchy. This makes it easier to grasp the capabilities of the different classes.

Figure 3.1 shows a hierarchy of the semantic models introduced in this chapter. The symbols Γ and Γ' denote *state spaces*. All programs map elements of Γ in some way to elements of Γ'. We permit the state spaces Γ and Γ' to be different because they are used in data refinement. This requires the ability to change state spaces. Elements of $\mathcal{F}(\Gamma, \Gamma')$ are called *state functions*. A state function models a deterministic program. State functions are embedded into *state relations*, denoted by $\mathcal{R}(\Gamma, \Gamma')$, and into *probabilistic state functions*, denoted by $\mathcal{M}(\Gamma, \Gamma')$. State relations extend state functions with nondeterminism. Probabilistic state functions are an extension of state functions where states are mapped randomly to succes-

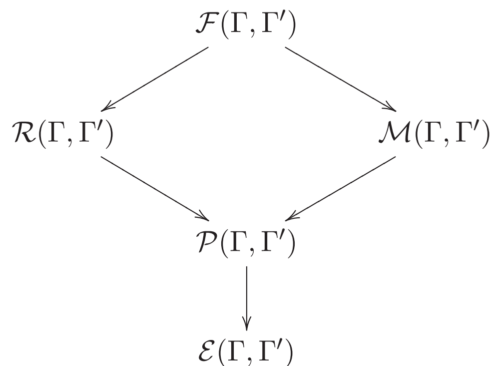


Figure 3.1: Semantic models

sor states. *Probabilistic state relations*, denoted by $\mathcal{P}(\Gamma, \Gamma')$, extend both state relations and probabilistic state functions. They model programs that exhibit nondeterminism and randomness. Probabilistic state relations are further embedded into *extended probabilistic state relations* $\mathcal{E}(\Gamma, \Gamma')$. In the extended model a transition from a state to another state is associated with an expected value. Note that these program notations are tailored to our needs and do not model divergence. See section 5.5 for a discussion of this matter.

We refer to any of the above simply as "program" if it is clear from the context which semantical model is meant. The remainder of this chapter treats the different semantic models. Section 3.1 introduces the concept of a state used throughout. It also introduces state functions. In section 3.2 state relations are defined, and all non-probabilistic program constructs are introduced. In section 3.3 probabilistic state functions are introduced, probabilistic state relations in section 3.4, and extended probabilistic state relations in section 3.5.

3.1 State Functions

A finite product Γ of given countable sets $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$,

$$\Gamma \;=\; \prod_{i=1}^{n} \Gamma_i, \quad n \geq 1 \;,$$

is called a *state space*. Using countable state spaces we can restrict ourselves to discrete probability theory later on. Their use is also customary in stochastic dynamic programming [110] which later chapters rely upon. Elements of a state space Γ are called *states*. A projection $\pi_i : \Gamma \to \Gamma_i$ is called a *variable* . By $\mathsf{var}.\Gamma$ we denote the set of variables of a state space Γ,

$$\mathsf{var}.\Gamma \;=\; \{\pi_1, \pi_2, \ldots, \pi_n\} \;.$$

A function $\phi : \Gamma \to \Gamma'$ is called a *state function*. The set of all state functions from Γ to Γ' is denoted by $\mathcal{F}(\Gamma, \Gamma')$. Sequential composition of state functions $\phi_1 \in \mathcal{F}(\Gamma, \Gamma')$ and $\phi_2 \in \mathcal{F}(\Gamma', \Gamma'')$ is defined by functional composition $\phi_1; \phi_2 \mathrel{\widehat{=}} \phi_2 \circ \phi_1$ By $\mathsf{id}_{\mathcal{F}}$ we denote the identity function. The subscript $\mathcal{F}$ denotes the model in which id is interpreted. As $\mathsf{id}_{\mathcal{F}}$ is embedded into other models the subscript is changed. In our notation skip is used as a synonym for id.

3.2 State Relations

State relations are capable of expressing nondeterminism. A state relation blocks execution if an attempt is made to execute it outside its domain. The

model does not support the notion of divergence present in the relational model of [4, 29, 38]. It is close to the one described in [33, 39] which is based on Z [112, 120]. The program notation is similar to the notation of [9], and in parts [2].

Let Γ and Γ' be state spaces. A relation $R : \Gamma \leftrightarrow \Gamma'$ is called a *state relation*. We denote the set of all state relations from Γ to Γ' by $\mathcal{R}(\Gamma, \Gamma')$.

For a predicate $q : \mathbb{P}\,\Gamma$ the guard $\mid q \mid \in \mathcal{R}(\Gamma)$ blocks execution in states where $(\neg\, q).\tau$ holds and behaves like skip otherwise. It is defined by

$$\mid q \mid.\tau.\tau' \;\;\widehat{=}\;\; \tau = \tau' \wedge q.\tau \ .$$

We also introduce the program stop which blocks any execution. It is defined by the false guard: $\mathsf{stop} \mathrel{\widehat{=}} \mid \mathsf{false} \mid$.

Nondeterministic choice between two state relations $R_1, R_2 \in \mathcal{R}(\Gamma, \Gamma')$ is defined by

$$(R_1 \sqcup R_2).\tau.\tau' \;\;\widehat{=}\;\; R_1.\tau.\tau' \vee R_2.\tau.\tau' \ .$$

For a set-valued expression $I : \Gamma \rightarrow \mathbb{P}\,\Delta$ finite nondeterministic choice between state relations $R_i \in \mathcal{R}(\Gamma, \Gamma')$, $i \in I.\tau$, is defined by

$$(\bigsqcup i : I \bullet R_i).\tau.\tau' \;\;\widehat{=}\;\; \exists\, i : I.\tau \bullet R_i.\tau.\tau' \ .$$

Finiteness of $\bigsqcup i : I \bullet R_i$ concerns the domain of I. We require that all sets $I.\tau$ are finite for all $\tau \in \Gamma$. Note that for $I.\tau = \varnothing$ for all τ finite nondeterministic choice behaves like stop. As with the requirement that state spaces be countable this one originates in stochastic dynamic programming. The known algorithms work only if there are at most finitely many alternatives in each state [98, 110]. In chapter 6 we present two such algorithms.

Sequential composition of state relation $R_1 \in \mathcal{R}(\Gamma, \Gamma')$ and state relation $R_2 \in \mathcal{R}(\Gamma', \Gamma'')$ is defined by

$$(R_1 ; R_2).\tau.\tau'' \;\;\widehat{=}\;\; \exists\, \tau' : \Gamma' \bullet R_1.\tau.\tau' \wedge R_2.\tau'.\tau'' \ .$$

Parallel composition of state relations $R_1 \in \mathcal{R}(\Gamma, \Gamma_1')$ and $R_2 \in \mathcal{R}(\Gamma, \Gamma_2')$ is defined by

$$(R_1 \parallel R_2).\tau.(\tau_1', \tau_2') \;\;\widehat{=}\;\; R_1.\tau.\tau_1' \wedge R_2.\tau.\tau_2' \ .$$

The resulting state relation maps states from the state space Γ shared by R_1 and R_2 to subsets of the product space $\Gamma_1' \times \Gamma_2'$. For a discussion of different parallel operators see [7].

For $\pi_i \in \mathsf{var}.\Gamma$, where $\Gamma = \prod_{i=1}^{n} \Gamma_i$, and an expression $e : \Gamma \rightarrow \Gamma_i$ the assignment of e to variable π_i, denoted by $\pi_i := e$, is defined by

$$(\pi_i := e).\tau.\tau' \;\;\widehat{=}\;\; \pi_i.\tau' = e.\tau \wedge (\forall\, j \bullet j \neq i \Rightarrow \pi_j.\tau' = \pi_j.\tau) \ .$$

The notion of expression we use is based on [9]. If $a \in \Gamma_i$ is a value we define the point-wise extension $\dot{a} : \Gamma \to \Gamma_i$ by $\dot{a} \mathrel{\widehat{=}} (\lambda \tau : \Gamma \bullet a)$. The assignment of a to π_i is then defined by

$$\pi_i := a \ \mathrel{\widehat{=}}\ \pi_i := \dot{a} \ .$$

To clarify the definition we give a small example of an assignment:

Example 3.1 Let $\mathbb{N} \times \mathbb{N}$ with variables $x = \pi_1$ and $y = \pi_2$. The expression $x + y$ has type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Let $(2, 3)$ be a state. Then

$$(x + y).(2, 3) \ = \ x.(2, 3) + y.(2, 3) \ = \ 2 + 3 \ = \ 5 \ ,$$

and

$$\begin{aligned} &(x := x + y).(2, 3).(5, 3) \\ &= \ x.(5, 3) = (x + y).(2, 3) \wedge y.(5, 3) = y.(2, 3) \\ &= \ x.(5, 3) = 5 \wedge y.(5, 3) = 3 \\ &= \ \mathsf{true} \ . \end{aligned}$$

We generalise from assignment to one variable to multiple assignment to a set of distinct variables. Multiple assignment is defined by parallel composition of assignments:

$$x_1, x_2 := e_1, e_2 \ \mathrel{\widehat{=}}\ x_1 := e_1 \parallel x_2 := e_2 \ .$$

We assume that the two assignments $x_1 := e_1$ and $x_2 := e_2$ have the proper types required in the parallel composition.

A state relation $R : \mathcal{R}(\Gamma, \Gamma')$ is called *deterministic* if $R.\tau.\tau_1' \wedge R.\tau.\tau_2'$ implies $\tau_1' = \tau_2'$ for all $\tau \in \Gamma$. State functions correspond to deterministic state relations. They are embedded into corresponding state relations by a function $\Uparrow : \mathcal{F}(\Gamma, \Gamma') \to \mathcal{R}(\Gamma, \Gamma')$, defined by,

$$(\Uparrow\phi).\tau.\tau' \ \mathrel{\widehat{=}}\ \phi.\tau = \tau' \ .$$

We occasionally write deterministic programs ϕ in the relational notation when we really mean state functions ϕ', given by $\phi = \Uparrow\phi'$. In such cases we explicitly refer to such a ϕ as a state function.

3.3 Probabilistic State Functions

In probability theory [80, 118] probabilistic state functions are generally referred to as stochastic matrices [72] or probability transition matrices [84]. They are used to model Markov chains [72]. Probabilistic state functions model probabilistic programs without further nondeterminism as present in state relations.

Probabilistic States

The state of probabilistic programs is described by a collection of states and their corresponding probabilities. More precisely, a *probabilistic state* $f : \Gamma \to \mathbb{R}_{\geq 0}$ is a function that assigns probabilities to states. The set of all probabilistic states over Γ is defined by

$$(\mathbb{D}\,\Gamma).f \;\;\widehat{=}\;\; \mathsf{card}.(\mathsf{car}.f) \in \mathbb{N} \wedge \textstyle\sum_{\tau:\Gamma} f.\tau = 1 \ ,$$

where $\mathsf{car}.f \subseteq \Gamma$, the *carrier* of f, is defined by

$$\mathsf{car}.f.\tau \;\;\widehat{=}\;\; f.\tau > 0 \ .$$

The set $\mathsf{car}.f$ describes a set of states in which a probabilistic program may be at some instant. The value $f.\tau$ is the probability that the program is in state τ. We use the notation $\tau \mathbin{@} p$ to represent $f.\tau = p$, and the notation

$$\{\tau_1 \mathbin{@} f.\tau_1, \tau_2 \mathbin{@} f.\tau_2, \ldots, \tau_n \mathbin{@} f.\tau_n\} \ ,$$

where $\mathsf{car}.f \subseteq \{\tau_1, \tau_2, \ldots, \tau_n\}$, to represent probabilistic state f itself. Similar notations are used in [109]. Probabilistic states are known as *densities*, or *masses*, in probability theory. We have decided to use the term 'probabilistic state' because phrases like 'program P is in probabilistic state f' sound more intuitive than if one of the other terms were used.

We define two operators on probabilistic states. We need addition and scalar product of functions $\Gamma \to \mathbb{R}_{\geq 0}$. They are defined by point-wise extension:

$$(f + g).\tau \;\;\widehat{=}\;\; f.\tau + g.\tau \ ,$$
$$(p * f).\tau \;\;\widehat{=}\;\; p * f.\tau \ ,$$

where $f, g \in \Gamma \to \mathbb{R}_{\geq 0}$ and $p \in \mathbb{R}_{\geq 0}$. For $p \in (0, 1)$ probabilistic addition of $f \in \mathbb{D}\,\Gamma$ and $g \in \mathbb{D}\,\Gamma$ is defined by

$$f \;{}_p\!\oplus\; g \;\;\widehat{=}\;\; p * f + (1 - p) * g \ .$$

If a program is in probabilistic state f with probability p and in probabilistic state g with probability $1 - p$, then its probabilistic state is $f \;{}_p\!\oplus\; g$. This situation arises when a program branches to probabilistic states f and g with the respective probabilities p and $1 - p$.

Example 3.2 Let $\Gamma = \mathbb{B}$. Let $f,\ g \in \mathbb{D}\,\mathbb{B}$ be probabilistic states,

$$f \;=\; \{\mathsf{true} \mathbin{@} \tfrac{3}{4}, \mathsf{false} \mathbin{@} \tfrac{1}{4}\} \ ,$$
$$g \;=\; \{\mathsf{true} \mathbin{@} \tfrac{1}{3}, \mathsf{false} \mathbin{@} \tfrac{2}{3}\} \ .$$

If a probabilistic state h of a program is f with probability $\frac{2}{5}$ and g with probability $\frac{3}{5}$, then it is in one of the states true or false with probability $\frac{1}{2}$ each.

$$\begin{aligned} & f \;{}_{\frac{2}{5}}\!\oplus\; g \\ = \;& \tfrac{2}{5} * f + \tfrac{3}{5} * g \\ = \;& \{\mathsf{true} \mathbin{@} \tfrac{6}{20}, \mathsf{false} \mathbin{@} \tfrac{2}{20}\} + \{\mathsf{true} \mathbin{@} \tfrac{3}{15}, \mathsf{false} \mathbin{@} \tfrac{6}{15}\} \\ = \;& \{\mathsf{true} \mathbin{@} \tfrac{1}{2}, \mathsf{false} \mathbin{@} \tfrac{1}{2}\} \ . \end{aligned}$$

For a finite set I and probabilities $p_i \in (0, 1)$, such that $\sum_{i:I} p_i = 1$, the probabilistic sum of the probabilistic states $f_i : \mathbb{D}\,\Gamma$ is defined by

$$\textstyle\bigoplus_{i:I} p_i \bullet f_i \;\;\widehat{=}\;\; \sum_{i:I} p_i * f_i \ .$$

As above, the sum $\sum_{i:I} p_i * f_i$ is defined by point-wise extension. Probabilistic addition and sum are indeed operators on probabilistic states as stated in the following proposition.

Proposition 3.3 If $p, p_i \in (0, 1)$ and $f, g, f_i \in \mathbb{D}\,\Gamma$, then $(f \;{}_p\!\oplus\; g) \in \mathbb{D}\,\Gamma$, and $(\bigoplus_{i:I} p_i \bullet f_i) \in \mathbb{D}\,\Gamma$.

In probability theory densities give rise to *probability distributions*. We set up a probability distribution $\mathbf{P}$ on a state space $\Gamma = \prod_{i=1}^{n} \Gamma_i$ by means of probabilistic states $f \in \mathbb{D}\,\Gamma$. We denote the product $\prod_{i=1}^{n} \pi_i$ of type $\Gamma \to \Gamma$ by π. So π is really the identity on Γ. We intend to use it as a variable name to refer to the value of a state. For $\Delta \subseteq \Gamma$ we define:

$$\mathbf{P}(\pi \in \Delta) \;\;\widehat{=}\;\; \textstyle\sum_{\tau:\Delta} f.\tau$$

Note that probabilistic state f is implicit on the left hand side. It is assumed to be associated with variable π. We write $\mathbf{P}(\pi = \tau)$ to denote $\mathbf{P}(\pi \in \{\tau\})$, which equals $f.\tau$. We define the *marginal* probability distribution of variables $\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_m}$ by

$$\mathbf{P}(\pi_{i_1} \in \Delta_{i_1}, \pi_{i_2} \in \Delta_{i_2}, \ldots, \pi_{i_m} \in \Delta_{i_m}) \;\;\widehat{=}\;\; \textstyle\sum_{\tau:\Delta} f.\tau \ ,$$

where Δ is the set $\{\tau : \Gamma \mid \pi_{i_1}.\tau \in \Delta_{i_1} \wedge \pi_{i_2}.\tau \in \Delta_{i_2} \wedge \ldots \wedge \pi_{i_m}.\tau \in \Delta_{i_m}\}$. We use the notation $\mathbf{P}(\pi_{i_1} = v_{i_1}, \pi_{i_2} = v_{i_2}, \ldots, \pi_{i_m} = v_{i_m})$ in the same way as $\mathbf{P}(\pi = \tau)$. In probability theory the variables π and π_i are called *random variables*. If $C : \Gamma \to \mathbb{R}_{\geq 0}$ is a *real* random variable, then

$$\mathbf{E}(C) \;\;\widehat{=}\;\; \textstyle\sum_{\tau:\Gamma} C.\tau * \mathbf{P}(\pi = \tau)$$

is called its *expectation*. Observe that $\mathbf{E}(C) \in \mathbb{R}_{\geq 0}$ exists because probabilistic states have a finite carrier. Defining the product of a probabilistic state $f : \mathbb{D}\,\Gamma$ and C by

$$f * C \;\;\widehat{=}\;\; \textstyle\sum_{\tau:\Gamma} f.\tau * C.\tau \ ,$$

the expectation $\mathbf{E}(C)$ can be written $f * C$.

Probabilistic Programs

A function $M : \Gamma \to \mathbb{D}\,\Gamma'$ is called a *probabilistic state function*. It takes an initial state to a final probabilistic state. The set of all probabilistic state functions from Γ to Γ' is denoted by $\mathcal{M}(\Gamma, \Gamma')$. The value $M.\tau.\tau'$ is the probability that next state is τ' under the condition that the present state is τ. In probability theory probabilistic state functions are called *conditional densities*. Conditional densities induce *conditional probability distributions*. Conditional probability distributions, usually denoted $\mathbf{P}(\pi' \in \Delta' \mid \pi = \tau)$, describe the probability that π' assumes a value in Δ' given that π equals τ. We define:

$$\mathbf{P}(\pi' \in \Delta' \mid \pi = \tau) \;\;\widehat{=}\;\; \textstyle\sum_{\tau':\Delta'} M.\tau.\tau' \; .$$

Thus π denotes the initial state and π' the successor state. Again, probabilistic state function M is implicit on the left hand side. The notation $\mathbf{P}(\pi' = \tau' \mid \pi = \tau)$ is used as before. Similarly, marginal conditional distributions are available. Based on conditional probability distributions, conditional expectations are defined. Let $C : \Gamma' \to \mathbb{R}_{\geq 0}$ be a real random variable. We define:

$$\mathbf{E}(C \mid \pi = \tau) \;\;\widehat{=}\;\; \textstyle\sum_{\tau':\Gamma'} C.\tau' * \mathbf{P}(\pi' \in \Delta' \mid \pi = \tau) \; .$$

Letting $(M * C).\tau \mathrel{\widehat{=}} (M.\tau) * C$, the conditional expectation $\mathbf{E}(C \mid \pi = \tau)$ equals the product $(M * C).\tau$.

Note that on finite state spaces $\Gamma = \{1, 2, \ldots, m\}$ and $\Gamma' = \{1, 2, \ldots, n\}$ probabilistic state functions M are usually represented as stochastic matrices

$$\begin{pmatrix} M.1.1 & M.1.2 & \cdots & M.1.n \\ M.2.1 & M.2.2 & \cdots & M.2.n \\ \vdots & \vdots & \ddots & \vdots \\ M.m.1 & M.m.2 & \cdots & M.m.n \end{pmatrix}$$

of type $(\Gamma \times \Gamma') \to \mathbb{R}_{\geq 0}$.

The product of a probabilistic state $f : \mathbb{D}\,\Gamma$ and a probabilistic state function $M \in \mathcal{M}(\Gamma, \Gamma')$ is defined component-wise by

$$(f * M).\tau' \;\;\widehat{=}\;\; \textstyle\sum_{\tau:\Gamma} f.\tau * M.\tau.\tau' \; .$$

State function M corresponds to a matrix. In the same manner the product $f * M$ corresponds to the product of a vector and a matrix.

$$\begin{pmatrix} f.1 & f.2 & \cdots & f.m \end{pmatrix} * \begin{pmatrix} M.1.1 & M.1.2 & \cdots & M.1.n \\ M.2.1 & M.2.2 & \cdots & M.2.n \\ \vdots & \vdots & \ddots & \vdots \\ M.m.1 & M.m.2 & \cdots & M.m.n \end{pmatrix}$$

$$= \begin{pmatrix} (f * M).1 & (f * M).2 & \cdots & (f * M).n \end{pmatrix}$$

Sequential composition of $M \in \mathcal{M}(\Gamma, \Gamma')$ and $N \in \mathcal{M}(\Gamma', \Gamma'')$ is defined by

$$(M; N).\tau.\tau'' \;\widehat{=}\; (M.\tau * N).\tau'' \;.$$

If Γ, Γ' and Γ'' are finite as above, then $M; N$ corresponds to the ordinary matrix product of M and N,

$$(M.\tau * N).\tau'' \;=\; \textstyle\sum_{\tau':\Gamma'} M.\tau.\tau' * N.\tau'.\tau'' \;.$$

State functions are embedded into probabilistic state functions by a function $\Uparrow : \mathcal{F}(\Gamma, \Gamma') \to \mathcal{M}(\Gamma, \Gamma')$, defined by

$$(\Uparrow\phi).\tau.\tau' \;\widehat{=}\; \chi.(\phi.\tau).\tau' \;,$$

where the characteristic function $\chi : \mathcal{M}(\Gamma)$ is defined by

$$\chi.\tau.\tau' \;\widehat{=}\; \begin{cases} 1 & \text{if } \tau = \tau' \\ 0 & \text{otherwise} \end{cases}$$

Using partial functions we express more general deterministic probabilistic programs: $\mathcal{D}(\Gamma, \Gamma') \,\widehat{=}\, \Gamma \nrightarrow \mathbb{D}\,\Gamma'$. We do not use partial probabilistic state functions with the concepts introduced in this section. They are a convenient means, though, to refer to deterministic programs.

Sequential Products

As a generalisation of sequential composition we introduce a finite sequential product of probabilistic state functions. Let $\Phi \in \mathsf{seq}\,\mathcal{M}(\Gamma)$ be a finite sequence of probabilistic state functions. The *sequential product* $\Pi.\Phi$ of sequence Φ is recursively defined by

$$\begin{aligned} \Pi.\langle\,\rangle \;&\widehat{=}\; \mathsf{id}_{\mathcal{M}} \;, \\ \Pi.(\Phi ^\frown \langle M \rangle) \;&\widehat{=}\; (\Pi.\Phi); M \;. \end{aligned}$$

Remember that $\mathsf{id}_{\mathcal{M}} \in \mathcal{M}(\Gamma)$ is the embedding of state function $\mathsf{id}_{\mathcal{F}} \in \mathcal{F}(\Gamma)$ into $\mathcal{M}(\Gamma)$. A discussion of the role of sequences $\Phi \in \mathsf{seq}\,\mathcal{M}(\Gamma)$ follows. Let k be the length of Φ. With each $\Phi.i$ we associate a conditional probability distribution

$$\mathbf{P}(\pi^i \in \Delta^i \mid \pi^{i-1} = \tau^{i-1}) \;\widehat{=}\; \textstyle\sum_{\tau^i \in \Delta^i} \Phi.i.\tau^{i-1}.\tau^i \;.$$

We use superscripts to indicate sequence ordering because subscripts are already in use. Probability distribution $\mathbf{P}$ induces a Markov chain [80], i.e.

$$\begin{aligned} &\mathbf{P}(\pi^k = \tau^k \mid \pi^{k-1} = \tau^{k-1}) \\ &\quad = \; \mathbf{P}(\pi^k = \tau^k \mid \pi^{k-1} = \tau^{k-1}, \ldots, \pi^0 = \tau^0) \;. \end{aligned}$$

Thus [80], the following property also holds:

$$\begin{aligned}&\mathbf{P}(\pi^k = \tau^k \mid \pi^0 = \tau^0)\\ &= \textstyle\sum_{\tau:\Gamma} \mathbf{P}(\pi^k = \tau^k \mid \pi^{k-1} = \tau) * \mathbf{P}(\pi^{k-1} = \tau \mid \pi^0 = \tau^0) \ .\end{aligned}$$

The expression $\mathbf{P}(\pi^k = \tau^k \mid \pi^0 = \tau^0)$ describes the probability of being in state τ^k after the whole sequence Φ has been executed, the initial state being τ^0. This is expressed by the following proposition:

Proposition 3.4 $\mathbf{P}(\pi^k = \tau^k \mid \pi^0 = \tau^0) = (\prod.\Phi).\tau^0.\tau^k$.

We prove proposition 3.4 by induction on $k \geq 0$:

$$\begin{aligned}&\mathbf{P}(\pi^0 = \tau^0 \mid \pi^0 = \tau^0)\\ &= 1\\ &= (\Pi.\langle\,\rangle).\tau^0.\tau^0 \ .\end{aligned}$$

And for $k > 0$:

$$\begin{aligned}&\mathbf{P}(\pi^k = \tau^k \mid \pi^0 = \tau^0)\\ &= \textstyle\sum_{\tau:\Gamma} \mathbf{P}(\pi^k = \tau^k \mid \pi^{k-1} = \tau) * \mathbf{P}(\pi^{k-1} = \tau \mid \pi^0 = \tau^0)\\ &= \textstyle\sum_{\tau:\Gamma} \Phi.k.\tau.\tau^k * (\Pi.(\Phi \uparrow k-1)).\tau^0.\tau\\ &= \textstyle\sum_{\tau:\Gamma} (\Pi.(\Phi \uparrow k-1)).\tau^0.\tau * \Phi.k.\tau.\tau^k\\ &= (\Pi.(\Phi \uparrow k-1); \Phi.k).\tau^0.\tau^k\\ &= (\Pi.\Phi).\tau^0.\tau^k \ .\end{aligned}$$

3.4 Probabilistic State Relations

State relations model nondeterminism by the multiplicity of final states. In probabilistic state relations introduced in this section this is replaced by the multiplicity of probabilistic states. They combine the expressiveness of state relations and probabilistic state functions. Technically, probabilistic state relations are defined similarly to the (probabilistic) relational model proposed in [48]. The difference is that in [48] nondeterminism is regarded as a generalised form of probabilistic choice. Probabilistic state relations keep both concepts separate from each other. They are based on Markov decision processes [16, 28].

A relation $P : \Gamma \leftrightarrow \mathbb{D}\,\Gamma'$ is called a *probabilistic state relation*. The set of all probabilistic state relations from Γ to Γ' is denoted by $\mathcal{P}(\Gamma, \Gamma')$.

State relations are embedded into probabilistic state relations by a function $\Uparrow : \mathcal{R}(\Gamma, \Gamma') \to \mathcal{P}(\Gamma, \Gamma')$, defined by

$$(\Uparrow R).\tau.f \ \ \widehat{=}\ \ \exists\, \tau' : R.\tau \bullet f = \chi.\tau' \ .$$

Remember that $(\Uparrow R).\tau$ is really a set which we write in functional notation. Thus $(\Uparrow R).\tau$ equals the set $\{\chi.\tau' \mid R.\tau.\tau'\}$.

The definition of nondeterministic choice between probabilistic state relations $P \in \mathcal{P}(\Gamma, \Gamma')$ and $Q \in \mathcal{P}(\Gamma, \Gamma')$ resembles that of nondeterministic choice between state relations. It is defined by

$$(P \sqcup Q).\tau.f \;\;\widehat{=}\;\; P.\tau.f \vee Q.\tau.f \;,$$

and finite nondeterministic choice between $Q_i \in \mathcal{P}(\Gamma, \Gamma')$, $i \in I.\tau$, by

$$(\bigsqcup i : I \bullet Q_i).\tau.f \;\;\widehat{=}\;\; \exists\, i : I.\tau \bullet Q_i.\tau.f \;.$$

For $p \in \Gamma \to (0,1)$ probabilistic choice $P\;{}_p\!\oplus\; Q$ between $P \in \mathcal{P}(\Gamma, \Gamma')$ and $Q \in \mathcal{P}(\Gamma, \Gamma')$ is defined arithmetically giving probability p to branch P and probability $1 - p$ to branch Q:

$$(P\;{}_p\!\oplus\; Q).\tau.h \;\;\widehat{=}\;\; \exists\, f : P.\tau,\; g : Q.\tau \bullet h = f\;{}_{p.\tau}\!\oplus\; g \;.$$

Let $I : \Gamma \to \mathbb{P}_1\,\Delta$ be a set-valued expression, all $I.\tau$ being finite sets. Also let $p_i \in \Gamma \to (0,1)$ for all $i \in I.\tau$, such that $\sum_{i:I.\tau} p_i.\tau = 1$. Finite probabilistic choice between programs $P_i \in \mathcal{P}(\Gamma, \Gamma')$ is defined by

$$\begin{aligned}(\bigoplus i : I \mid p_i \bullet P_i).\tau.h \;\;\widehat{=}\;\; & \exists\, F \bullet (\forall\, i : I.\tau \bullet F.i \in P_i.\tau) \wedge \\ & h = \textstyle\bigoplus_{i:I.\tau} p_i.\tau \bullet F.i \;.\end{aligned}$$

We demonstrate the use of finite probabilistic choice with a simple program.

Example 3.5 Let $\Gamma = 1\,..\,6$, $x = \pi_1$. Also let $P \in \mathcal{P}(\Gamma)$,

$$P \;\;=\;\; \bigoplus m : 1\,..\,6 \mid \tfrac{1}{6} \bullet x := m \;.$$

Let $\tau \in \Gamma$ be some state. For $x := m \in \mathcal{R}(\Gamma)$, $(x := m).\tau = \{m\}$. Embedded into $\mathcal{P}(\Gamma)$ it becomes

$$(x := m).\tau \;\;=\;\; \{\chi.m\} \;.$$

Hence $P.\tau$ equals the singleton set $\{\bigoplus_{m:1..6} \frac{1}{6} \bullet \chi.m\}$ consisting of the probabilistic state

$$\{1 \mathbin{@} \tfrac{1}{6}, 2 \mathbin{@} \tfrac{1}{6}, 3 \mathbin{@} \tfrac{1}{6}, 4 \mathbin{@} \tfrac{1}{6}, 5 \mathbin{@} \tfrac{1}{6}, 6 \mathbin{@} \tfrac{1}{6}\} \;.$$

So execution of program P corresponds to rolling a fair die, i.e. a die where all outcomes have equal probability.

Parallel composition of $P_1 \in \mathcal{P}(\Gamma, \Gamma_1')$ and $P_2 \in \mathcal{P}(\Gamma, \Gamma_2')$ is defined by

$$(P_1 \parallel P_2).\tau.h \;\;\widehat{=}\;\; \exists\, f : P_1.\tau,\; g : P_2.\tau \bullet h = f \parallel g \;,$$

where $f \parallel g$ denotes the point-wise product of functions f and g:

$$(f \parallel g).(\tau_1, \tau_2) \;\widehat{=}\; f.\tau_1 * g.\tau_2 \;.$$

A parallel composition $P_1 \parallel P_2$ relates states of state space Γ with probabilistic states of type $\mathbb{D}(\Gamma_1' \times \Gamma_2')$. It is easy to prove that the point-wise product of two probabilistic states is a probabilistic state:

Proposition 3.6 If $f_1 \in \mathbb{D}\,\Gamma_1$ and $f_2 \in \mathbb{D}\,\Gamma_2$, then $f_1 \parallel f_2 \in \mathbb{D}\,(\Gamma_1 \times \Gamma_2)$.

In probability theory $f_1 \parallel f_2$ is referred to as a *joint* density of π_1 and π_2. It induces a *joint probability distribution* as well. Let $\mathbf{P}_1$ and $\mathbf{P}_2$ be the probability distributions associated with f_1 and f_2 respectively. The joint probability distribution $\mathbf{P}$ of π_1 and π_2 corresponding to $f_1 \parallel f_2$ is given by:

$$\begin{aligned}
&\mathbf{P}(\pi_1 \in \Delta_1, \pi_2 \in \Delta_2) \\
&\quad= \textstyle\sum_{\tau_1 \in \Delta_1} \sum_{\tau_2 \in \Delta_2} (f_1 \parallel f_2).(\tau_1, \tau_2) \\
&\quad= \textstyle\sum_{\tau_1 \in \Delta_1} \sum_{\tau_2 \in \Delta_2} f_1.\tau_1 * f_2.\tau_2 \\
&\quad= \textstyle\sum_{\tau_1 \in \Delta_1} f_1.\tau_1 * \sum_{\tau_2 \in \Delta_2} f_2.\tau_2 \\
&\quad= \mathbf{P}_1(\pi_1 \in \Delta_1) * \mathbf{P}_2(\pi_2 \in \Delta_2) \;.
\end{aligned}$$

In probability theory two random variables are called *independent* if their joint distribution is the product of their marginal probability distributions. Hence $\mathbf{P}(\pi_1 \in \Delta_1, \pi_2 \in \Delta_2)$ is the distribution of two independent variables. This agrees with the view that the two programs P_1 and P_2 do not affect each other when executed in parallel. In general, a probability distribution over any product space is called a joint distribution. It may not be made up from independent variables though.

Sequential composition of probabilistic state relations $P \in \mathcal{P}(\Gamma, \Gamma')$ and $Q \in \mathcal{P}(\Gamma', \Gamma'')$ is denoted by $P; Q$. It is executed by picking all probabilistic states f from $P.\tau$ in turn for each initial state τ. Then the weighted average of the probabilistic states $M.\tau' \in Q.\tau'$ for all $\tau' \in \mathsf{car}.f$ is taken the weights being the probabilities $f.\tau'$. That weighted average is just the product $f * M$. The product $f * M$ can be interpreted as the expected state that is reached if each probabilistic state $M.\tau$ is chosen with probability $f.\tau$:

$$f * M \;=\; \textstyle\bigoplus_{\tau : \mathsf{car}.f} f.\tau \bullet M.\tau \;. \qquad (3.1)$$

The state functions M are taken from the set $\mathsf{fun}.Q \subseteq \mathsf{dom}.Q \to \mathbb{D}\,\Gamma$ which is defined by

$$\mathsf{fun}.Q.M \;\widehat{=}\; \forall \tau : \mathsf{dom}.Q \bullet M.\tau \in Q.\tau \;.$$

The state functions $M \in \mathsf{fun}.Q$ describe the deterministic implementations of a probabilistic state relation. If there is an intermediate state $\tau' \in \mathsf{car}.f$ which Q cannot map anywhere, execution of probabilistic state f is blocked.

We note that Q can map each state from $\mathsf{car}.f$ to some probabilistic state iff $\mathsf{car}.f \subseteq \mathsf{dom}.Q$, and define:

$$(P; Q).\tau.g \;\widehat{=}\; \exists f : P.\tau,\ M : \mathsf{fun}.Q \bullet \mathsf{car}.f \subseteq \mathsf{dom}.Q \wedge g = f * M \ .$$

Two examples on sequential composition follow. The first one demonstrates sequential composition using the matrix notation of section 3.3. The second one presents a blocked execution.

Example 3.7 Let $P \in \mathcal{P}(\{0\}, \{1,2\})$ and $Q \in \mathcal{P}(\{1,2\}, \{1,2,3\})$,

$$\begin{aligned} P.0 &= \left\{\{1 \mathbin{@} \tfrac{1}{2}, 2 \mathbin{@} \tfrac{1}{2}\}\right\} \ , \\ Q.i &= \left\{\{i \mathbin{@} \tfrac{1}{3}, (i+1) \mathbin{@} \tfrac{2}{3}\}\right\} \ , \text{ for } i \in \{1,2\}. \end{aligned}$$

We calculate $P; Q$. The set $\mathsf{fun}.Q$ equals $\{M\}$, where

$$M = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} & 0 \\ 0 & \frac{1}{3} & \frac{2}{3} \end{pmatrix} \ ,$$

and $P.0$ equals $\{(\,\frac{1}{2}\ \frac{1}{2}\,)\}$. The product $(\,\frac{1}{2}\ \frac{1}{2}\,) * M$ equals $(\,\frac{1}{6}\ \frac{1}{2}\ \frac{1}{3}\,)$. Hence $P; Q \in \mathcal{P}(\{0\}, \{1,2,3\})$ is given by

$$(P; Q).0 = \left\{\{1 \mathbin{@} \tfrac{1}{6}, 2 \mathbin{@} \tfrac{1}{2}, 3 \mathbin{@} \tfrac{1}{3}\}\right\} \ .$$

Example 3.8 Let $P \in \mathcal{P}(\{0\}, \{1,2,3\})$ and $Q \in \mathcal{P}(\{1,2\}, \{0\})$,

$$\begin{aligned} P.0 &= \left\{\{1 \mathbin{@} \tfrac{1}{3}, 2 \mathbin{@} \tfrac{1}{3}, 3 \mathbin{@} \tfrac{1}{3}\}\right\} \ , \\ Q.i &= \left\{\{0 \mathbin{@} 1\}\right\} \ , \text{ for } i \in \{1,2\}. \end{aligned}$$

Program Q blocks the execution of $\{1 \mathbin{@} \frac{1}{3}, 2 \mathbin{@} \frac{1}{3}, 3 \mathbin{@} \frac{1}{3}\}$, because $3 \in \mathsf{car}.\{1 \mathbin{@} \frac{1}{3}, 2 \mathbin{@} \frac{1}{3}, 3 \mathbin{@} \frac{1}{3}\}$ but $3 \notin \mathsf{dom}.Q$. Hence,

$$(P; Q).0 = \varnothing \ .$$

A finite iteration P^n of a probabilistic state relation P executes it n times. Iteration of $P \in \mathcal{P}(\Gamma)$ is defined by

$$\begin{aligned} P^0 &\;\widehat{=}\; \mathsf{id}_{\mathcal{P}} \ , \\ P^{n+1} &\;\widehat{=}\; P^n; P \ . \end{aligned}$$

We note that iterates of certain probabilistic state relations are equivalent to the finite nondeterministic choice over corresponding products of state functions. We need an embedding $\Uparrow : \mathcal{D}(\Gamma, \Gamma') \to \mathcal{P}(\Gamma, \Gamma')$ of state functions into probabilistic state relations to express this. It is defined by:

$$(\Uparrow M).\tau.f \;\widehat{=}\; \tau \in \mathsf{dom}.M \wedge f = M.\tau \ .$$

The definition is also easily applied to total probabilistic state functions $M \in \mathcal{M}(\Gamma, \Gamma')$.

Proposition 3.9 Let $P \in \mathcal{P}(\Gamma)$ be a probabilistic state relation. Let $n \in \mathbb{N}$, and assume that $\mathsf{dom}.P = \Gamma$. Then:

$$P^n \;=\; \bigsqcup \Phi : SEQ \bullet \Pi.\Phi \ ,$$

where $SEQ = \{\Phi \mid \mathsf{card}.\Phi = n \wedge (\forall\, i : 1 \,..\, n \bullet \Phi.i \in \mathsf{fun}.P)\}$.

Note that the condition $\mathsf{dom}.P = \Gamma$ means that P does not contain any probabilistic state that might be blocked during execution.

3.5 Extended Probabilistic State Relations

Probabilistic state relations map states to sets of probabilistic states. This models the presence of nondeterminism and probability at the same time. In this section probabilistic state relations are augmented with costs. A cost statement is included in the program notation. Upon encountering such a statement a program incurs the specified cost, so that an execution yields an expected cost as well as a final probabilistic state. In [109] expectation transformers are introduced where expected costs entirely replace probabilistic state. This approach is not appropriate for our purposes. Extended probabilistic state relations are used to describe the behaviour of probabilistic action systems in chapter 4.

A *cost* is a nonnegative real number. Similarly to probabilistic addition and sum in section 3.3 we introduce such operators for costs. They yield expected costs when choices between different program branches occur probabilistically. Let $p \in (0, 1)$ and $c_1,\ c_2 \in \mathbb{R}_{\geq 0}$. We define:

$$c_1 \ {}_p\!\oplus\ c_2 \ \ \widehat{=}\ \ p * c_1 + (1 - p) * c_2 \ .$$

The expression on the right hand side calculates the expected cost incurred after a probabilistic choice has been made. If cost c_1 is incurred with probability p and cost c_2 with probability $1 - p$, then the expected cost incurred is $c_1 \ {}_p\!\oplus\ c_2$. We refer to both, $c_1 \ {}_p\!\oplus\ c_2$ and $\bigoplus_{i:I} p_i \bullet c_i$, as expected costs. The latter is defined by:

$$\textstyle\bigoplus_{i \in I} p_i \bullet c_i \ \ \widehat{=}\ \ \sum_{i:I} p_i * c_i \ ,$$

where I is a finite set, $c_i \in \mathbb{R}_{\geq 0}$, and $p_i \in (0, 1)$ such that $\sum_{i \in I} p_i = 1$. Expected costs are expectations with the state space Γ replaced by a set of execution branches I. Let probability distribution $\mathbf{P}$ be given by $\mathbf{P}(\mathit{branch} = i) = p_i$, and $C : I \to \mathbb{R}_{\geq 0}$, $C.i = c_i$, be a cost function. These two give rise to an expectation $\mathbf{E}(C)$ corresponding to the expected cost above:

$$\mathbf{E}(C) \;=\; \textstyle\bigoplus_{i:I} p_i \bullet c_i \ .$$

Extended probabilistic state relations relate states with pairs of expected costs and probabilistic states (c, f). Costs c have no effect on the execution

of a program. They record expected costs associated with execution paths. Considered on their own expected costs yield a more abstract description of the behaviour of a program. If f is chosen as the successor probabilistic state, then expected cost c is incurred by the program. Cost c does characterise the choice made but without details about the probabilistic state chosen. The reason for referring to c itself as expected cost is that it is usually not specified directly. Instead, costs are specified at different locations in a program. From these an expected cost c is derived.

A relation $P : \Gamma \leftrightarrow (\mathbb{R}_{\geq 0} \times \mathbb{D}\,\Gamma')$ is called an *extended probabilistic state relation*. The set of all these is denoted by $\mathcal{E}(\Gamma, \Gamma')$. Probabilistic state relations correspond to their extensions with all costs zero. This is expressed by embedding function $\Uparrow : \mathcal{P}(\Gamma, \Gamma') \to \mathcal{E}(\Gamma, \Gamma')$, defined by

$$(\Uparrow P).\tau.(c, f) \;\;\widehat{=}\;\; c = 0 \wedge P.\tau.f \ .$$

Arbitrary nonnegative costs are associated with programs by means of cost statements. For a real expression $e : \Gamma \to \mathbb{R}_{\geq 0}$ the cost statement $|\ e\ | : \mathcal{E}(\Gamma)$ costs state τ at value $e.\tau$ and behaves like skip on the state. It is defined by

$$|\ e\ |.\tau.(c, f) \;\;\widehat{=}\;\; c = e.\tau \wedge f = \chi.\tau \ .$$

Costs can be abstracted by function $\Downarrow : \mathcal{E}(\Gamma, \Gamma') \to \mathcal{P}(\Gamma, \Gamma')$, defined by,

$$(\Downarrow P).\tau.f \;\;\widehat{=}\;\; \exists\, c : \mathbb{R}_{\geq 0} \bullet P.\tau.(c, f) \ .$$

The remaining program constructs are defined analogously to those of probabilistic state relations. They additionally describe the calculation of expected costs associated with a program.

Nondeterministic choice between extended probabilistic state relations $P \in \mathcal{E}(\Gamma, \Gamma')$ and $Q \in \mathcal{E}(\Gamma, \Gamma')$ is defined by

$$(P \sqcup Q).\tau.(c, f) \;\;\widehat{=}\;\; P.\tau.(c, f) \vee Q.\tau.(c, f) \ .$$

Nondeterministic choice now takes costs into account that are associated with the alternatives P and Q. Different costs are incurred by a program depending on which alternative an execution follows. Finite nondeterministic choice is defined similarly.

For $p \in \Gamma \to (0, 1)$ probabilistic choice between $P \in \mathcal{E}(\Gamma, \Gamma')$ and $Q \in \mathcal{E}(\Gamma, \Gamma')$ is defined by

$$\begin{aligned}(P \;{}_p\!\oplus\; Q).\tau.(e, h) \;\;\widehat{=}\;\; & \exists (c, f) : P.\tau,\ (d, g) : Q.\tau \bullet \\ & \qquad e = c \;{}_{p.\tau}\!\oplus\; d \wedge h = f \;{}_{p.\tau}\!\oplus\; g \ .\end{aligned}$$

In words, (e, h) consists of an expected cost e and a corresponding expected probabilistic state h resulting from probabilistic choice $P \;{}_p\!\oplus\; Q$ in state τ. We illustrate the use of probabilistic choice and cost statement in a small example:

Example 3.10 Let $\Gamma = \mathbb{N}$ be a state space, $x = \pi_1$, and

$$P \;=\; |\, x \,| \;{}_{\frac{1}{2}}\!\oplus\; |\, 2 * x \,| \;.$$

We calculate the meaning of P as

$$\begin{aligned}
& P.n.(e,h) \\
=\;& \exists (c,f) : P.\tau,\ (d,g) : Q.\tau \bullet e = c \;{}_{\frac{1}{2}}\!\oplus\; d \wedge h = f \;{}_{\frac{1}{2}}\!\oplus\; g \\
=\;& e = n \;{}_{\frac{1}{2}}\!\oplus\; (2 * n) \wedge h = \chi.n \;{}_{\frac{1}{2}}\!\oplus\; \chi.n \\
=\;& e = \tfrac{1}{2} * n + n \wedge h = \chi.n \\
=\;& e = \tfrac{3}{2} * n \wedge h = \chi.n \;.
\end{aligned}$$

Hence the expected cost of an execution of P in state n is $\frac{3}{2} * n$.

Finite probabilistic choice between extended probabilistic state relations is defined as well:

$$\begin{aligned}
(\textstyle\bigoplus i : I \mid p_i \bullet P_i).\tau.(e,h) \;\widehat{=}\;& \exists\, C,\, F \bullet \\
& (\forall\, i : I.\tau \bullet (C.i, F.i) \in P_i.\tau) \wedge \\
& e = \textstyle\bigoplus_{i:I.\tau} p_i.\tau \bullet C.i \wedge \\
& h = \textstyle\bigoplus_{i:I.\tau} p_i.\tau \bullet F.i \;,
\end{aligned}$$

where $I : \Gamma \to \mathbb{P}_1\, \Delta$ is a set-valued expression, $P_i \in \mathcal{E}(\Gamma, \Gamma')$ are programs, and $p_i \in \Gamma \to (0,1)$ probabilities such that $\sum_{i:I.\tau} p_i.\tau = 1$.

As before we need the deterministic implementations of program to define sequential composition. Let $P \in \mathcal{E}(\Gamma, \Gamma')$. An implementation consists of a probabilistic state function M and a cost function C. Cost function C specifies the costs $C.\tau$ program M incurs given an initial state $\tau \in \mathsf{dom}.M$. Function fun which maps P to its deterministic implementations (C, M) is defined by

$$\mathsf{fun}.P.(C,M) \;\widehat{=}\; \forall\, \tau : \mathsf{dom}.P \bullet (C.\tau, M.\tau) \in P.\tau \;.$$

Hence the set $\mathsf{fun}.P$ is a subset of $(\mathsf{dom}.P \to \mathbb{R}_{\geq 0}) \times (\mathsf{dom}.P \to \Gamma')$. We define sequential composition of $P \in \mathcal{P}(\Gamma, \Gamma')$ and $Q \in \mathcal{P}(\Gamma', \Gamma'')$ by

$$\begin{aligned}
(P;Q).\tau.(d,g) \;\widehat{=}\;& \exists (c,f) : P.\tau,\ (C,M) : \mathsf{fun}.Q \bullet \mathsf{car}.f \subseteq \mathsf{dom}.Q \wedge \\
& d = c + f * C \wedge g = f * M \;.
\end{aligned}$$

The expression $c + f * C$ is the expected cost associated a sequential execution of P and Q. Cost c is incurred when f is chosen as the successor probabilistic state of τ. It corresponds to an expectation $\mathbf{E}(c)$. The expected cost incurred afterwards depends on the intermediate probabilistic state f. It equals $f * C$ where the costs $C.\tau'$ associated with $M.\tau'$ are

weighed according to the probability $f.\tau'$, that τ' occurs as an intermediate state. Remember that $f * C = \bigoplus_{\tau:\mathsf{car}.f} f.\tau \bullet C.\tau$ denotes an expectation $\mathbf{E}(C)$. Hence the overall expected cost is $\mathbf{E}(c) + \mathbf{E}(C) = c + f * C$.

We also define a lifted form of parallel composition of programs $P \in \mathcal{E}(\Gamma, \Gamma'_1)$ and $Q \in \mathcal{E}(\Gamma, \Gamma'_2)$. By adding the costs of the component programs P and Q we keep the correspondence $P \parallel Q = P; Q$ between certain programs P and Q in the presence of costs. We define

$$\begin{aligned}(P \parallel Q).\tau.(e, h) \;\widehat{=}\; & \exists (c, f) : P.\tau,\, (d, g) : Q.\tau \bullet \\ & e = c + d \wedge h = f \parallel g \;.\end{aligned}$$

The type of $P \parallel Q$ is $\mathcal{E}(\Gamma, \Gamma'_1 \times \Gamma'_2)$. We illustrate the motivation behind this definition, in particular the choice of $c + d$ as compound cost, by an example.

Example 3.11 Let $\Gamma = \mathbb{N} \times \mathbb{N}$ be a state space, $x = \pi_1$ and $y = \pi_2$. Consider the programs $P = |\, x \,|; |\, y \,|$ and $Q = |\, x \,| \parallel |\, y \,|$. Assume $|\, x \,|$ and $|\, y \,|$ have proper types in both cases, so that $P,\, Q \in \mathcal{E}(\Gamma)$. The equality $P = Q$ holds because:

$$\begin{aligned}& (e, h) \in P.\tau \\ \Leftrightarrow\;& e = x.\tau + \chi.\tau * y \wedge h = \chi.\tau * \mathsf{id}_{\mathcal{M}} \\ \Leftrightarrow\;& e = x.\tau + \chi.\tau.\tau * y.\tau \wedge h = \chi.(\tau_1, \tau_2) \\ \Leftrightarrow\;& e = x.\tau + y.\tau \wedge h = \chi.\tau_1 \parallel \chi.\tau_2 \\ \Leftrightarrow\;& (e, h) \in Q.\tau \;,\end{aligned}$$

where we have assumed that $\tau = (\tau_1, \tau_2)$. Programs P and Q both incur costs x and y with probability 1. So the expected cost incurred is $x + y$.

3.6 Algebraic Laws

This section presents algebraic properties of the programming language defined in the preceding sections. Each algebraic law has a number (Ln) for reference. Most of the laws stated are used in proofs in later chapters. They are grouped into different categories to improve readability. Most of the laws are stated as equations. Some are stated as set inclusions however. Set inclusion is used as the natural ordering of programs as customary in relational semantics [9]. It is easy to see that all embedding functions $\Uparrow$ are homomorphisms with respect to the operators shared between the different program classes. All laws are states for the largest program class possible but are evidently valid for smaller classes as well. Let P, P_1, P_2, ... denote extended probabilistic state relations throughout.

Sequential Composition

Sequential composition is associative. It has skip as its unit and stop as its zero.

$$\mathsf{skip}; P \;=\; P \;=\; P; \mathsf{skip} \tag{L1}$$

$$\mathsf{stop}; P \;=\; \mathsf{stop} \;=\; P; \mathsf{stop} \tag{L2}$$

$$P_1; (P_2; P_3) \;=\; (P_1; P_2); P_3 \tag{L3}$$

Validity of law (L2) depends on the fact that P does not model nontermination. It only models blocking, and stop is the most blocking program.

Parallel Composition

Strictly speaking, law (L4) does not hold. However, we always assume that variables of program are ordered according to the projections π_i. This could be achieved formally but would complicate the semantics unnecessarily. So parallel composition is commutative under a suitable isomorphism. Parallel composition has skip as its unit. Note however, that this is also an equation that relies on a suitable isomorphism because the type of P changes. We generally assume that variables that are not assigned a value remain unchanged. Effectively this means parallel composition with a skip statement of appropriate type.

$$P_1 \parallel P_2 \;=\; P_2 \parallel P_1 \tag{L4}$$

$$P \parallel \mathsf{skip} \;=\; P \tag{L5}$$

Nondeterministic Choice

Nondeterministic choice is idempotent, commutative, and has stop as its unit.

$$P \;=\; P \sqcup P \tag{L6}$$

$$P_1 \sqcup P_2 \;=\; P_2 \sqcup P_1 \tag{L7}$$

$$P \;=\; P \sqcup \mathsf{stop} \tag{L8}$$

Sequential composition distributes through nondeterministic choice from the left if the left component is not probabilistic. It distributes through nondeterministic choice from the right in the general case though. Also parallel composition distributes through nondeterministic choice from both sides. Let R be a state relation:

$$R; (P_1 \sqcup P_2) \;=\; (R; P_1) \sqcup (R; P_2) \tag{L9}$$

$$P; (P_1 \sqcup P_2) \;\supseteq\; (P; P_1) \sqcup (P; P_2) \tag{L10}$$

$$(P_1 \sqcup P_2); P \;=\; (P_1; P) \sqcup (P_2; P) \tag{L11}$$

$$P \parallel (P_1 \sqcup P_2) \;=\; (P \parallel P_1) \sqcup (P \parallel P_2) \tag{L12}$$

$$(P_1 \sqcup P_2) \parallel P \;=\; (P_1 \parallel P) \sqcup (P_2 \parallel P) \tag{L13}$$

Law (L10) because the single programs P_1 and P_2 may be more blocking than the joint program $P_1 \sqcup P_2$. This can be seen from the following example:

Example 3.12 Let $P, |\, x\,|, |\, \neg\, x\,| \in \mathcal{P}(\mathbb{B})$, $P \;=\; (x := \mathsf{true}\ {}_{\frac{1}{2}}\!\oplus\ x := \mathsf{false})$. Then $P.\mathsf{true} = P.\mathsf{false} = \{f\}$ where $f = \{\mathsf{true} @ \frac{1}{2}, \mathsf{false} @ \frac{1}{2}\}$. Consequently, $\mathsf{car}.f = \{\mathsf{true}, \mathsf{false}\}$. Hence, because $\mathsf{dom}.|\, x\,| = \{\mathsf{true}\}$,

$$P; |\, x\,| \;=\; \mathsf{stop}.$$

Similarly, $P; |\, \neg\, x\,| = \mathsf{stop}$. Hence, $(P; |\, x\,|) \sqcup (P; |\, \neg\, x\,|) = \mathsf{stop}$. However, since $|\, x\,| \sqcup |\, \neg\, x\,| = \mathsf{skip}$,

$$\mathsf{dom}.(|\, x\,| \sqcup |\, \neg\, x\,|) \;=\; \mathsf{dom}.(\mathsf{skip}) \;=\; \{\mathsf{true}, \mathsf{false}\}\ ,$$

thus, $P; (|\, x\,| \sqcup |\, \neg\, x\,|) = P \supset \mathsf{stop} = (P; |\, x\,|) \sqcup (P; |\, \neg\, x\,|)$.

Probabilistic Choice

Probabilistic choice is idempotent and skew-commutative. Let $p : \Gamma \to \mathbb{R}_{\geq 0}$ be probabilities with $p.\tau \in (0, 1)$:

$$P \;=\; P\ {}_p\!\oplus\ P \tag{L14}$$

$$P_1\ {}_p\!\oplus\ P_2 \;=\; P_2\ {}_{(1-p)}\!\oplus\ P_1 \tag{L15}$$

Let M be a probabilistic state function. Sequential composition distributes M through probabilistic choice from the right. It distributes M from left if the branching probability equals a constant $c \in \mathbb{R}_{\geq 0}$. It distributes through probabilistic choice from the right . Parallel composition distributes probabilistic state function M through probabilistic choice from both sides.

$$M; (P_1\ {}_c\!\oplus\ P_2) \;=\; (M; P_1)\ {}_c\!\oplus\ (M; P_2) \tag{L16}$$

$$(P_1\ {}_p\!\oplus\ P_2); M \;=\; (P_1; M)\ {}_p\!\oplus\ (P_2; M) \tag{L17}$$

$$M \parallel (P_1\ {}_p\!\oplus\ P_2) \;=\; (M \parallel P_1)\ {}_p\!\oplus\ (M \parallel P_2) \tag{L18}$$

$$(P_1\ {}_p\!\oplus\ P_2) \parallel M \;=\; (P_1 \parallel M)\ {}_p\!\oplus\ (P_2 \parallel M) \tag{L19}$$

If instead of probabilistic state function M we use a program P that may contain nondeterminism, the following weaker results hold:

$$P; (P_1\ {}_c\!\oplus\ P_2) \;\subseteq\; (P; P_1)\ {}_c\!\oplus\ (P; P_2) \tag{L20}$$

$$(P_1\ {}_p\!\oplus\ P_2); P \;\subseteq\; (P_1; P)\ {}_p\!\oplus\ (P_2; P) \tag{L21}$$

$$P \parallel (P_1\ {}_p\!\oplus\ P_2) \;\subseteq\; (P \parallel P_1)\ {}_p\!\oplus\ (P \parallel P_2) \tag{L22}$$

Law (L20) is similar to law (L10). On the right hand side of the equation P may unblock some behaviour of P_1 and P_2. Here is an example:

Example 3.13 Let $P, \mid x \mid, \mid \neg\, x \mid \in \mathcal{P}(\mathbb{B})$, $P = (x := \mathsf{true} \sqcup x := \mathsf{false})$. Then,

$$P; (\mid x \mid {}_{\frac{1}{2}}\!\oplus \mid \neg\, x \mid) \;\subset\; (P; \mid x \mid) {}_{\frac{1}{2}}\!\oplus (P; \mid \neg\, x \mid) \; . \tag{3.2}$$

The two programs $\mid x \mid$ and $\mid \neg\, x \mid$ have disjoint domains, $\mathsf{dom}.\mid x \mid = \{\mathsf{true}\}$ and $\mathsf{dom}.\mid \neg\, x \mid = \{\mathsf{false}\}$, implying that $\mid x \mid {}_{\frac{1}{2}}\!\oplus \mid \neg\, x \mid = \mathsf{stop}$, and

$$P; (\mid x \mid {}_{\frac{1}{2}}\!\oplus \mid \neg\, x \mid) \;=\; \mathsf{stop} \; .$$

For $b \in \{\mathsf{true}, \mathsf{false}\}$:

$$\begin{aligned}(P; \mid x \mid).b \;&=\; \Big\{\{\mathsf{true} \mathbin{@} 1\}\Big\} \; , \\ (P; \mid \neg\, x \mid).b \;&=\; \Big\{\{\mathsf{false} \mathbin{@} 1\}\Big\} \; ,\end{aligned}$$

implying

$$((P; \mid x \mid) {}_{\frac{1}{2}}\!\oplus (P; \mid \neg\, x \mid)).b \;=\; \Big\{\{\mathsf{true} \mathbin{@} \tfrac{1}{2}, \mathsf{false} \mathbin{@} \tfrac{1}{2}\}\Big\} \; , \text{thus (3.2).}$$

In law (L21) on the right hand side P may increase the nondeterminism inside the probabilistic choice. This may cause an increase in probabilistic alternatives as well. See the following example.

Example 3.14 Let $P, \mathsf{skip}, (x := \mathsf{true}) \in \mathcal{P}(\mathbb{B})$, $P = (\mathsf{skip} \sqcup x := \mathsf{false})$. Then

$$(\mathsf{skip} {}_{\frac{1}{2}}\!\oplus x := \mathsf{true}); P \;\subset\; (\mathsf{skip}; P) {}_{\frac{1}{2}}\!\oplus (x := \mathsf{true}; P) \; . \tag{3.3}$$

For program $Q = (\mathsf{skip} {}_{\frac{1}{2}}\!\oplus x := \mathsf{true})$:

$$\begin{aligned}Q.\mathsf{true} \;&=\; \Big\{\{\mathsf{true} \mathbin{@} 1\}\Big\} \; , \\ Q.\mathsf{false} \;&=\; \Big\{\{\mathsf{true} \mathbin{@} \tfrac{1}{2}, \mathsf{false} \mathbin{@} \tfrac{1}{2}\}\Big\} \; .\end{aligned}$$

Hence for the left hand side of (3.3):

$$\begin{aligned}(Q; P).\mathsf{true} \;&=\; Q.\mathsf{true} \cup \Big\{\{\mathsf{false} \mathbin{@} 1\}\Big\} \; , \\ (Q; P).\mathsf{false} \;&=\; Q.\mathsf{false} \cup \Big\{\{\mathsf{false} \mathbin{@} 1\}\Big\} \; .\end{aligned}$$

For the programs $\mathsf{skip}; P$ and $x := \mathsf{true}; P$ on the right hand side of (3.3):

$$\begin{aligned}(\mathsf{skip}; P).\mathsf{true} \;&=\; \Big\{\{\mathsf{true} \mathbin{@} 1\}, \{\mathsf{false} \mathbin{@} 1\}\Big\} \; , \\ (\mathsf{skip}; P).\mathsf{false} \;&=\; \Big\{\{\mathsf{false} \mathbin{@} 1\}\Big\} \; ,\end{aligned}$$

and for $b \in \{\mathsf{true}, \mathsf{false}\}$:

$$(x := \mathsf{true}; P).b \;=\; \Big\{\{\mathsf{true} \mathbin{@} 1\}, \{\mathsf{false} \mathbin{@} 1\}\Big\} \; .$$

Thus, with $R = (\mathsf{skip}; P) \ {}_{\frac{1}{2}}\!\oplus\ (x := \mathsf{true}; P)$:

$$\begin{aligned} R.\mathsf{true} &= (Q;P).\mathsf{true} \cup \Big\{\{\mathsf{true} \ @\ \tfrac{1}{2}, \mathsf{false} \ @\ \tfrac{1}{2}\}\Big\} \ , \\ R.\mathsf{false} &= (Q;P).\mathsf{false} \ . \end{aligned}$$

A similar effect as observed in the last example occurs if sequential composition is replaced by parallel composition in law (L22):

Example 3.15 Let $(x := k), (y := k) \in \mathcal{P}(\{1,2\} \times \{1,2\}, \{1,2\})$ for $k \in \{1,2\}$. Assume $x = \pi_1$, $y = \pi_2$. It holds:

$$P \parallel (y := 1 \ {}_{\frac{1}{2}}\!\oplus\ y := 2) \ \subset \ (P \parallel y := 1) \ {}_{\frac{1}{2}}\!\oplus\ (P \parallel y := 2) \ , \tag{3.4}$$

where $P = (x := 1 \sqcup x := 2)$. On the left hand side of (3.4):

$$(y := 1 \ {}_{\frac{1}{2}}\!\oplus\ y := 2).n \ = \ \Big\{\{1 \ @\ \tfrac{1}{2}, 2 \ @\ \tfrac{1}{2}\}\Big\}$$

for $n \in \{1,2\} \times \{1,2\}$. And:

$$\begin{aligned} &(P \parallel (y := 1 \ {}_{\frac{1}{2}}\!\oplus\ y := 2)).n \\ &\quad = \ \Big\{\{(1,1) \ @\ \tfrac{1}{2}, (1,2) \ @\ \tfrac{1}{2}\}, \{(2,1) \ @\ \tfrac{1}{2}, (2,2) \ @\ \tfrac{1}{2}\}\Big\} \ . \end{aligned}$$

For the left hand side of (3.4):

$$(P \parallel y := k).n \ = \ \Big\{\{(1,k) \ @\ 1\}, \{(2,k) \ @\ 1\}\Big\} \ .$$

Finally,

$$\begin{aligned} &(P \parallel y := 1) \ {}_{\frac{1}{2}}\!\oplus\ (P \parallel y := 2) \\ &\quad = \ \Big\{\{(1,1) \ @\ \tfrac{1}{2}, (1,2) \ @\ \tfrac{1}{2}\}, \{(2,1) \ @\ \tfrac{1}{2}, (2,2) \ @\ \tfrac{1}{2}\}\Big\} \cup \\ &\qquad\ \ \Big\{\{(1,1) \ @\ \tfrac{1}{2}, (2,2) \ @\ \tfrac{1}{2}\}, \{(2,1) \ @\ \tfrac{1}{2}, (1,2) \ @\ \tfrac{1}{2}\}\Big\} \ . \end{aligned}$$

Nondeterministic Choice and Probabilistic Choice

Probabilistic choice distributes through nondeterministic choice but not vice versa. Let $p \in \Gamma \to (0,1)$:

$$P \ {}_p\!\oplus\ (P_1 \sqcup P_2) \ = \ (P \ {}_p\!\oplus\ P_1) \sqcup (P \ {}_p\!\oplus\ P_2) \tag{L23}$$

$$P \sqcup (P_1 \ {}_p\!\oplus\ P_2) \ \subseteq \ (P \sqcup P_1) \ {}_p\!\oplus\ (P \sqcup P_2) \tag{L24}$$

In law (L24) the probabilistic choice on the right hand side of the equation is offered more alternatives to choose from. See the following example:

Example 3.16 Let $\mathsf{skip}, P_1, P_2 \in \mathcal{P}(\mathbb{B})$,

$$P_1 \;=\; (|\; x \;|; x := \mathsf{false}) \;,\; P_2 \;=\; (|\; \neg\, x \;|; x := \mathsf{true}) \;.$$

Then

$$\mathsf{skip} \sqcup (P_1 \;{}_{\frac{1}{2}}\!\oplus\; P_2) \;\subset\; (\mathsf{skip} \sqcup P_1) \;{}_{\frac{1}{2}}\!\oplus\; (\mathsf{skip} \sqcup P_2) \;. \tag{3.5}$$

Regarding the left hand side of (3.5) we have

$$(|\; x \;|; x := \mathsf{false}) \;{}_{\frac{1}{2}}\!\oplus\; (|\; \neg\, x \;|; x := \mathsf{true}) \;=\; \mathsf{stop} \;,$$

and also $\mathsf{skip} \sqcup \mathsf{stop} = \mathsf{skip}$. Now consider the two component programs $\mathsf{skip} \sqcup P_1$ and $\mathsf{skip} \sqcup P_2$ on the right hand side of (3.5):

$$\begin{aligned}(\mathsf{skip} \sqcup P_1).\mathsf{true} &= \Big\{\{\mathsf{true} \;@\; 1\}, \{\mathsf{false} \;@\; 1\}\Big\} \;,\\ (\mathsf{skip} \sqcup P_1).\mathsf{false} &= \Big\{\{\mathsf{false} \;@\; 1\}\Big\} \;,\end{aligned}$$

and

$$\begin{aligned}(\mathsf{skip} \sqcup P_2).\mathsf{true} &= \Big\{\{\mathsf{true} \;@\; 1\}\Big\} \;,\\ (\mathsf{skip} \sqcup P_2).\mathsf{false} &= \Big\{\{\mathsf{true} \;@\; 1\}, \{\mathsf{false} \;@\; 1\}\Big\} \;.\end{aligned}$$

Finally, for the entire program $Q = (\mathsf{skip} \sqcup P_1) \;{}_{\frac{1}{2}}\!\oplus\; (\mathsf{skip} \sqcup P_2)$ on the right hand side of (3.5):

$$\begin{aligned}Q.\mathsf{true} &= \Big\{\{\mathsf{true} \;@\; 1\}, \{\mathsf{true} \;@\; \tfrac{1}{2}, \mathsf{false} \;@\; \tfrac{1}{2}\}\Big\} \;,\\ Q.\mathsf{false} &= \Big\{\{\mathsf{false} \;@\; 1\}, \{\mathsf{true} \;@\; \tfrac{1}{2}, \mathsf{false} \;@\; \tfrac{1}{2}\}\Big\} \;.\end{aligned}$$

It holds $\mathsf{skip} \subset Q$, and thus the claimed inequality.

Assignment

If a variable is assigned to itself, nothing changes. Also, consecutive assignments to the same variable can be merged. Law (L27) holds only if variable y does not occur in expression e_1. Let x and y be distinct variables:

$$x := x \;=\; \mathsf{skip} \tag{L25}$$

$$x := e_1; x := e_2 \;=\; x := [x := e_1]\, e_2 \tag{L26}$$

$$x := e_1; y := e_2 \;=\; y := [x := e_1]\, e_2; x := e_1 \tag{L27}$$

Guard statement and assignment are quasi-commutative with respect to sequential composition. The same holds for cost statement and assignment.

$$x := e; |\; q \;| \;=\; |\; [x := e]\, q \;|; x := e \tag{L28}$$

$$x := e; |\; r \;| \;=\; |\; [x := e]\, r \;|; x := e \tag{L29}$$

If two assignments do not affect each other, their sequential composition equals their parallel composition. Law (L30) holds only if variable x does not occur in expression e_2.

$$x := e_1; y := e_2 \;=\; x := e_1 \parallel y := e_2 \tag{L30}$$

Assignment quasi-distributes through finite nondeterministic choice with respect to sequential composition.

$$x := e; (\bigsqcup i : I \bullet P_i) \;=\; \bigsqcup i : ([x := e]\, I) \bullet (x := e; P_i) \tag{L31}$$

Assignment is also quasi-distributive through both kinds of probabilistic choice with respect to sequential composition.

$$x := e; (P_1 \;{}_p\!\oplus\; P_2) \;=\; (x := e; P_1) \;{}_{[x:=e]\,p}\!\oplus\; (x := e; P_2) \tag{L32}$$

$$x := e; (\bigoplus i : I \mid p_i \bullet P_i) \tag{L33}$$
$$= \;\bigoplus i : ([x := e]\, I) \mid ([x := e]\, p_i) \bullet (x := e; P_i)$$

Guard and Cost Statement

The true guard and zero cost statements have no effect. Two consecutive guard statements can be conjoined, and two consecutive cost statements summed. Also, expected costs can be calculated prior to probabilistic choices.

$$\lfloor \mathsf{true} \rfloor \;=\; \mathsf{skip} \tag{L34}$$
$$\lfloor q_1 \rfloor; \lfloor q_2 \rfloor \;=\; \lfloor q_1 \wedge q_2 \rfloor \tag{L35}$$
$$\lfloor 0 \rfloor \;=\; \mathsf{skip} \tag{L36}$$
$$\lfloor r_1 \rfloor; \lfloor r_2 \rfloor \;=\; \lfloor r_1 + r_2 \rfloor \tag{L37}$$
$$(\lfloor r_1 \rfloor; P_1) \;{}_p\!\oplus\; (\lfloor r_2 \rfloor; P_2) \;=\; \lfloor r_1 \;{}_p\!\oplus\; r_2 \rfloor; (P_1 \;{}_p\!\oplus\; P_2) \tag{L38}$$

3.7 Remarks

We have decided to realise nondeterministic choice $\sqcup$ and probabilistic choice ${}_p\!\oplus$ as incomparable notions. The order of programs in all relational models only concerns nondeterministic choice. This is in contrast to the approaches of [48, 88] where nondeterministic choice $\oplus$ is defined as a kind of generalised probabilistic choice,

$$Q \oplus R \;\widehat{=}\; \bigcup_{p \in [0,1]} Q \;{}_p\!\oplus\; R\ .$$

With this model nondeterministic choice $\oplus$ and probabilistic choice ${}_p\!\oplus$ are related to each other, $Q \oplus R \supseteq Q \;{}_p\!\oplus\; R$. In the corresponding refinement notion $\oplus$ can be refined by ${}_p\!\oplus$ with the understanding that probabilistic

choice is more deterministic. Using this approach one can derive bounds for the probability that one program refines another.

We have chosen $\sqcup$ as nondeterministic choice because in the refinement of probabilistic action systems (see chapter 4) we seek a non-probabilistic control program that is derived by refining an initial nondeterministic system. This control program can then be regarded as an implementation of a standard action system as presented in chapter 2.4. The operator $\sqcup$ models construction time nondeterminism [48] as opposed to runtime nondeterminism of [88]. In this respect our model is closer to the kind of nondeterminism proposed in [77], or the second model presented in [48], which is derived from [64].

Additionally, the models in [48, 88] support a notion of divergence. The dynamic programming approach for performance analysis taken in this work (see chapter 5) cannot handle divergent systems at all. Hence, divergence is not part of the model. Our definitions of sequential composition and probabilistic choice, however, are very similar to the corresponding definitions in model one in [48].

In the definition of $P\ {}_p\!\oplus\ Q$ we could have allowed $p \in \{0, 1\}$ as well, as in [48]. We decided not to do so for the following reasons. Equation (3.1) treats states that have zero probability as if they cannot occur whereas $P\ {}_p\!\oplus\ Q$ would not:

$$\mathsf{stop}\ {}_0\!\oplus\ \mathsf{skip} \quad = \quad \mathsf{stop}\ .$$

Although branch stop has zero probability it is treated as if it could occur. As a consequence it blocks the execution of the entire statement. It would be possible to change the semantics to achieve the validity of the algebraic law $P\ {}_0\!\oplus\ Q = Q$. But this would complicate the semantics without obtaining any gain. The law would permit us to remove programs that are chosen with zero probability, or introduce them. It does not make sense to specify a program just to remove it later. In [48] values $p = 0$ or $p = 1$ make sense because of the employed refinement relation.

Chapter 4

Probabilistic Action Systems

The behaviour of machines is described by traces of actions, i.e. action names and parameter values. This corresponds to the view that there is some environment in which the machine operates. That environment synchronises with the machine to use its functionality. To that end the environment can observe the traces of actions the machine can engage in. Refinement of machines is defined correspondingly as a relationship between traces of actions, the idea being that a machine operating in some environment ought to be replaceable with a refined, or rather, improved machine. To achieve this it is sufficient that the initial machine is able to simulate the behaviour of the refined machine.

In this chapter probabilistic action systems are introduced. Behaviour and refinement of probabilistic action systems differ fundamentally from behaviour and refinement of machines. Their behaviour is described by traces of costs. A cost is a non-negative real number. Actions themselves are not observable. Consequently, the kind of synchronisation and cooperation present in action systems is not possible. In fact a probabilistic action system models a fully synchronised system which includes the environment. The traces of a probabilistic action system are manifestations of its cost structure. The cost structure is specified by way of the cost statement. Refinement in this context means subsumption of the cost structure of a refined probabilistic action system by that of the abstract probabilistic action system. In chapter 5 refinement is linked to a performance measure. Refinement is otherwise independent from any specific performance measure. This is discussed in section 5.5.

Finally, probabilistic action systems also have a different time model than action systems. Both use discrete time, but action systems model time as the order in which actions occur. Looking at a trace one can say that some action occurs after another one. Nothing is said about the temporal difference between the two actions though. A cost trace of a probabilistic action system conceals the actions. One only knows that a cost trace corresponds to a

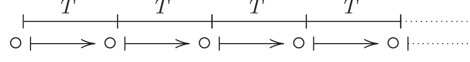


Figure 4.1: Discrete time model of probabilistic action systems

history of state changes. In addition, it is assumed that each state change takes exactly one unit of time. The amount of time that makes one unit of time depends on the application context. Figure 4.1 depicts the used time model. A unit of time is an time interval of length T. We also refer to the time interval as *time slot* or *transition period*. The state change occurring during a transition period is termed *transition*.

4.1 Behaviour of Probabilistic Action Systems

A probabilistic action system $\mathbf{A}$, or *system* for short, is defined by a tuple (Γ, I, P) where

Γ is a state space,

$I \subseteq \mathbb{D}\,\Gamma$ is a set of probabilistic states, the *initialisation* of $\mathbf{A}$, and

$P \in \mathcal{E}(\Gamma)$ is a program, the *action* of $\mathbf{A}$.

The behaviour of a probabilistic action system $\mathbf{A}$ is described by its traces and impasses. A trace is a sequence of non-negative real numbers. The real numbers correspond to expected costs system $\mathbf{A}$ may incur during single transitions periods in its evolution. An impasse is a trace after which the system may not be able to continue. The state of a probabilistic action system is not observable directly. However, the traces of a system are the observable consequence of state changes the system undergoes.

At any time the state of $\mathbf{A}$ is known up to some probability. Hence the evolution of $\mathbf{A}$ can be described via sequences of probabilistic states and their associated costs. The sequences of probabilistic states are implicitly contained in the definition of $\mathsf{path}.\mathbf{A}$. If $\mathsf{path}.\mathbf{A}.t.f$ is true, action system $\mathbf{A}$ may undergo trace t leading to probabilistic state f.

$$\begin{aligned}\mathsf{path}.\mathbf{A}.\langle\,\rangle.f \;&\widehat{=}\; f \in I\\ \mathsf{path}.\mathbf{A}.t^\frown\langle c\rangle.f \;&\widehat{=}\; \exists\, g \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (c, f) \in g * P\ .\end{aligned}$$

The term $g * P$ denotes the set $\{(g * C, g * M) \mid (C, M) \in \mathsf{fun}.P\}$ if $\mathsf{car}.g \subseteq \mathsf{dom}.P$, and $\varnothing$ otherwise. The semantics of $\mathbf{A}$ abstracts from the state of the system. It is defined by $\mathsf{beh}.\mathbf{A} \mathrel{\widehat{=}} (\mathsf{tr}.\mathbf{A}, \mathsf{im}.\mathbf{A})$ where $\mathsf{tr}.\mathbf{A}$ are called the *traces* of $\mathbf{A}$,

$$\mathsf{tr}.\mathbf{A}.t \;\widehat{=}\; \exists f \bullet \mathsf{path}.\mathbf{A}.t.f\ ,$$

```
system SYS
constants
   C; ...
constraints
   predicate_C;
sets
   S = expression_S; ...
variables
   v_1 : expression_{v_1};
     ⋮
   v_n : expression_{v_n};
initialisation
   program_ι;
programs
   p = program_p; ...
actions
   a_1 = program_{a_1};
     ⋮
   a_m = program_{a_m};
end
```

Figure 4.2: Syntax of probabilistic action systems

and $\mathsf{im}.\mathbf{A}$ are called the *impasses* of $\mathbf{A}$

$$\mathsf{im}.\mathbf{A}.t \ \widehat{=} \ \exists f \bullet \mathsf{path}.\mathbf{A}.t.f \wedge f * P = \varnothing \ .$$

4.2 Syntactic Representation

The syntactic representation of probabilistic action systems is based on a subset of B and the guarded command language of [9] adding probabilistic imperative features as in [48, 88].

A syntactic system has the structure pictured in figure 4.2. In the constants section natural number constants are declared. Usually these have the function of parameters for a specification because only finite state spaces are supported. Possible values of the constants are constrained by the predicate of the constraints section. The sets section contains constant sets that are used elsewhere in the specification. The state is declared as a collection of variables in the variables section. The data types that can be used are similar to those of B: finite sets, Cartesian products, power sets, functions, and relations. See chapter 2 and appendix B for details on types and available mathematical notation. Initial values for variables are given in the initialisation section. The operational behaviour is further specified in the

actions section containing a number of actions. Initialisation and actions of a system are atomic. Reoccuring program text can be given a name and declared in the programs section. All available program constructs are defined in chapter 3.

The semantics of a syntactic system is a probabilistic action system $\mathbf{SYS} = (\Gamma, I, P)$. The state space Γ is the Cartesian product of the types of the variables v_1, v_2, ..., v_n:

$$\Gamma \;=\; \mathit{expression}_{v_1} \times \mathit{expression}_{v_2} \times \ldots \times \mathit{expression}_{v_n} \;.$$

The initialisation I of $\mathbf{SYS}$ is derived from the initialisation section:

$$I \;=\; \mathsf{ran}.(\mathit{program}_\iota) \;.$$

As with action systems $\mathit{program}_\iota$ must not assume that there is a state before its execution. This implies that $\mathsf{ran}.(\mathit{program}_\iota)$ is a finite set. The action of $\mathbf{SYS}$ is the finite choice over all named actions a_1, a_2, ..., a_m of SYS:

$$P \;=\; \bigsqcup a : \{a_1, a_2, \ldots, a_m\} \bullet \mathit{program}_a \;.$$

The names themselves are only used for reference and have no semantical significance. We usually refer to syntactical systems as probabilistic action systems, thus, blurring the difference between syntax and semantics. This is never a problem since it is always apparent from the notation what is meant.

4.3 Example: Polling System

The cyclic polling system treated in this section is similar to those investigated in [52, 106, 110]. However, they use continuous time whereas we use a discrete time model. The system is specified as shown in figure 4.3. It consists of a number of $STATIONS$ arranged in a ring. Each station is equipped with a buffer of some maximal $CAPACITY$. The stations are numbered from 1. The successor of station i is station $i + 1$ unless i is the last station whose successor is station 1. Function $NEXT$ defines the successor relation. A server travels around the ring from one station to its successor. Being at station i the server can either remain there or move to the next station, where it must reside for at least one unit of time. If the server decides to stay at some location i it serves packets from $\mathit{buffer}.i$ with an average rate of $\frac{1}{4}$ packets per unit of time, or it idles if the buffer is empty. Packets arrive at the beginning of each time slot with a rate of $\frac{1}{10}$. On arrival at station i a packet is added to $\mathit{buffer}.i$ if that buffer is not full. The mean time it takes for the server to get from one station to the next is 2 units of time. This means the probability of arriving at the next station after one unit of time is $\frac{1}{2}$, since the arrival process is geometric.

system $POLLING$
constants
  $STATIONS$; $CAPACITY$;
constraints
  $STATIONS > 0 \wedge CAPACITY > 0$;
sets
  $STATION \ = \ 1 \,..\, STATIONS$;
  $NEXT \ = \ (\lambda\, s : 1 \,..\, (STATIONS - 1) \bullet s + 1) \cup \{STATIONS \mapsto 1\}$;
  $ARRIVAL \ = \ \{\mathsf{true} \mapsto 0.1, \mathsf{false} \mapsto 0.9\}$;
variables
  $station \ : \ STATION$;
  $buffer \ : \ STATION \to 0 \,..\, CAPACITY$;
  $moving \ : \ \mathbb{B}$;
programs
  $arrival \ = $ // arrival of packets at stations
    $\bigoplus S : \mathbb{P}(buffer^{\sim}[0 \,..\, CAPACITY - 1])$
      $\mid (\prod s : buffer^{\sim}[0 \,..\, CAPACITY - 1] \bullet ARRIVAL.(s \in S)) \bullet$
        $buffer := buffer \Lleftarrow (\lambda\, s : S \bullet buffer.s + 1)$;
  $departure \ = $ // departure of processed packets from $station$
    $\mid buffer.station > 0 \mid$;
      $(buffer := buffer \Lleftarrow \{station \mapsto buffer.station - 1\} \ {}_{0.25}\!\oplus \ \mathsf{skip})$
    $\sqcup$
    $\mid buffer.station = 0 \mid$;
initialisation
  $station := 1 \parallel moving := \mathsf{false} \parallel buffer := STATION \times \{0\}$;
actions
  $serve \ =$
    $\mid \sum s : STATION \bullet buffer.s \mid$;
    $\mid \neg\, moving \mid; arrival; departure$;
  $walk \ =$
    $\mid 1.0 \mid$;
    $\mid \sum s : STATION \bullet buffer.s \mid$;
    $arrival$;
    $(moving := \mathsf{true} \ {}_{0.5}\!\oplus \ (moving := \mathsf{false} \parallel station := NEXT.station))$;
end

Figure 4.3: A cyclic polling system

The state of system *POLLING* is described by three variables. Variable *station* contains the location of the server. If the boolean variable *moving* has value true the server is travelling between two stations. The total function *buffer* holds the number of packets waiting at each station. Initially, the server is at station 1 and stationary. Also, all buffers are empty.

In the programs section the joint *arrival* process and the *departure* process are defined. The joint arrival process is the product of the *ARRIVAL* processes of all non-full buffers. Let $W = buffer^{\sim}[0..CAPACITY - 1]$ be the set of stations having space available in their buffer, and $S \subseteq W$ a set of stations at which packets arrive. Then the probability of arrivals at exactly all stations in S is $\prod s : W \bullet ARRIVAL.(s \in S)$, which equals

$$\left(\frac{1}{10}\right)^{\mathsf{card}.S} * \left(\frac{9}{10}\right)^{\mathsf{card}.(W-S)} .$$

A departure, i.e. service completion, takes place with probability $\frac{1}{4}$ if the buffer at station *station* (the location of the server) *buffer.station* is non-empty. Otherwise the server idles.

There are two actions *serve* and *walk* which represent the two tasks of the server. It can either service a station or move to the next station, the choice between the two actions being nondeterministic. At the beginning of each time slot arrivals are dealt with. Remember that execution of an action represents what may happen in one unit of time. The actual servicing of packets in action *serve* is described by program *departure*. If the server is moving it arrives with probability $\frac{1}{2}$ at the next station, and *moving* is set to false. With probability $\frac{1}{2}$ it continues moving towards that station.

Costs are specified as real numbers which are not part of the language elsewhere. For each packet waiting in any queue a cost of 1.0 is incurred, in total,

$$\sum s : STATION \bullet buffer.s \ .$$

The system also incurs a cost of 1.0 per unit of time for moving between stations.

4.4 Cost Refinement

Refinement of probabilistic action systems preserves cost structure and reduces the number of possible impasses. The cost structure of a system is the basis for the performance measures introduced in chapter 5. The performance measures are only defined for live systems, i.e. systems that have no impasses. Since the number of impasses cannot increase a measure defined on a live system is also defined on any of its refinements. Hence the performance of a system can be compared to that of any of its refinements.

Probabilistic action system $\mathbf{C}$ refines probabilistic action system $\mathbf{A}$, denoted by $\mathbf{A} \sqsubseteq \mathbf{C}$, if all behaviour possible for $\mathbf{C}$ is also possible for $\mathbf{A}$. That is system $\mathbf{C}$ has less traces and less impasses than system $\mathbf{A}$. We define:

$$\mathbf{A} \sqsubseteq \mathbf{C} \;\;\widehat{=}\;\; \mathsf{tr}.\mathbf{C} \subseteq \mathsf{tr}.\mathbf{A} \wedge \mathsf{im}.\mathbf{C} \subseteq \mathsf{im}.\mathbf{A} \ .$$

Probabilistic action systems $\mathbf{A}$ and $\mathbf{C}$ are called equivalent, denoted by $\mathbf{A} \equiv \mathbf{C}$, if $\mathbf{A} \sqsubseteq \mathbf{C}$ and $\mathbf{C} \sqsubseteq \mathbf{A}$. A probabilistic action system $\mathbf{A}$ is called *live*, if it has no impasses,

$$\mathsf{live}.\mathbf{A} \;\;\widehat{=}\;\; \mathsf{im}.\mathbf{A} = \varnothing \ .$$

Finite traces that can be continued indefinitely are called infinite traces. The infinite traces $\mathsf{itr}.\mathbf{A} \subseteq \mathrm{seq}_\infty \mathbb{R}_{\geq 0}$ of a system $\mathbf{A}$ are defined by:

$$\mathsf{itr}.\mathbf{A}.v \;\;\widehat{=}\;\; \forall\, t : \mathsf{seq}\, \mathbb{R}_{\geq 0} \bullet t \leq v \Rightarrow \mathsf{tr}.\mathbf{A}.t \ .$$

The behaviour of a live system is entirely described by the infinite traces it can engage in:

Proposition 4.1 Let $\mathbf{A}$ be a live system. Then for all $t \in \mathsf{seq}\, \mathbb{R}_{\geq 0}$:

$$\mathsf{tr}.\mathbf{A}.t \;\;\Leftrightarrow\;\; \exists\, v : \mathsf{itr}.\mathbf{A} \bullet t \leq v \ .$$

Clearly, $\mathbf{A} \sqsubseteq \mathbf{C}$ implies $\mathsf{itr}.\mathbf{A} \supseteq \mathsf{itr}.\mathbf{C}$. Also, if $\mathbf{A}$ has no impasses and is refined by $\mathbf{C}$, then $\mathbf{C}$ is live. If both systems are live, cost refinement can be expressed solely by inclusion of infinite traces. The following property holds:

Proposition 4.2 Let $\mathbf{A}$ and $\mathbf{C}$ be two live probabilistic action systems. Then:

$$\mathbf{A} \sqsubseteq \mathbf{C} \;\;\Leftrightarrow\;\; \mathsf{itr}.\mathbf{C} \subseteq \mathsf{itr}.\mathbf{A} \ .$$

An alternative way to characterise a system as live is by means of the states it can reach. For a probabilistic action system $\mathbf{A} = (\Gamma, I, P)$ we define the set of reachable states, $\mathsf{reach}.\mathbf{A}$, by

$$\mathsf{reach}.\mathbf{A} \;\;\widehat{=}\;\; \bigcup \{\mathsf{car}.f \mid (\exists\, n : \mathbb{N} \bullet f \in I; (\Downarrow P)^n)\} \ .$$

The set $\mathsf{reach}.\mathbf{A}$ contains all states a system can possibly reach, starting from a state with initial probability greater than zero, by some iteration of its action P. We note:

Proposition 4.3 Let $\mathbf{A} = (\Gamma, I, P)$. Then:

$$\mathsf{live}.\mathbf{A} \;\;\Leftrightarrow\;\; \mathsf{reach}.\mathbf{A} \subseteq \mathsf{dom}.P \ .$$

Proposition 4.3 states that a system does not contain impasses if and only if its action can continue from any reachable state.

4.4.1 Simulation

When proving cost refinements $\mathbf{A} \sqsubseteq \mathbf{C}$ we do not use the definition directly. Instead of checking entire traces we compare the step by step behaviour of systems **A** and **C**. Simulation is a proof technique to do this: System **C** refines system **A** if system **A** can simulate the behaviour of system **C** step by step. This method is widely used in formalisms that have trace-based behaviour [8, 67, 119] and in data refinement [2, 87, 120].

Although the behaviour of probabilistic action systems is modelled by traces of costs instead of traces of actions, simulation of probabilistic action systems looks very similar to simulation of machines. Theorem 4.4 presents a simulation method that establishes refinement. In the next section two more variants of simulation are presented that establish equivalence between two systems.

Theorem 4.4 Let $\mathbf{A} = (\Gamma_A, I, P)$ and $\mathbf{C} = (\Gamma_C, J, Q)$ be two probabilistic action systems. If there is a probabilistic state function $M \in \mathcal{D}(\Gamma_A, \Gamma_C)$, such that

$$J \;\subseteq\; I; M \tag{PS1}$$

$$\mathsf{dom}.P \;\subseteq\; \mathsf{dom}.(M; Q) \tag{PS2}$$

$$M; Q \;\subseteq\; P; M \tag{PS3}$$

then $\mathbf{A} \sqsubseteq \mathbf{C}$.

We call the probabilistic state function M in theorem 4.4 *probabilistic simulation*. It creates a link between the states of the two systems.

Condition (PS1) ensures that the initialisation of the concrete system **C** can be matched by the initialisation of the abstract system **A**. Condition (PS2) ensures that the abstract system can refuse to continue whenever the concrete system can do so. Hence any impasse of the concrete system must also be an impasse of the abstract system. Condition (PS3) ensures that the effect of the concrete action is matched by that of the abstract action. Note, that (PS3) also establishes the required connection between the cost structures of the two systems.

It appears a strong requirement for the probabilistic simulation M to be deterministic. Nondeterministic simulations would allow us to introduce impasses as can be seen from example 4.5.

Example 4.5 Figure 4.4 shows two probabilistic action systems **A** and **C**. It holds $\mathsf{im}.\mathbf{A} = \varnothing$ and $\mathsf{im}.\mathbf{C} = \mathsf{seq}(\{0\})$. Their traces both equal $\mathsf{seq}(\{0\})$. Consequently **C** does not refine **A**. Using the nondeterministic simulation

$$R \;=\; C.wieder$$

we are able to prove $\mathbf{A} \sqsubseteq \mathbf{C}$. We have to prove conditions (PS1) to (PS3):

system A	system C
variables	variables
$x \; : \; \mathbb{B};$	$x \; : \; \mathbb{B};$
initialisation	initialisation
$x := \mathsf{true};$	$x := \mathsf{true};$
actions	actions
$immer \;=\; \mid x \mid;$	$wieder \;=\; \mid x \mid; (x := \mathsf{true} \sqcup x := \mathsf{false});$
end	end

Figure 4.4: **C** does not refine **A**

(PS1) $C.\mathsf{initialisation} \subseteq (x := \mathsf{true} \sqcup x := \mathsf{false}) = A.\mathsf{initialisation}; R.$

(PS2) Because $(R; C.wieder) = C.wieder$ and $\mathsf{dom}.(C.wieder) = \{\mathsf{true}\}$,

$$\mathsf{dom}.(A.immer) \;=\; \{\mathsf{true}\} \;\subseteq\; R; C.wieder \;.$$

(PS3) It also holds $(A.immer; R) = C.wieder$, so with the above,

$$R; C.wieder \;\subseteq\; A.immer; R \;.$$

This would prove $\mathbf{A} \sqsubseteq \mathbf{C}$.

The following example 4.6 demonstrates the use of probabilistic simulation. It provides an abstraction technique often used in queueing theory: the individuals waiting in a queue are represented by a number that corresponds to the size of the queue when they arrive.

Example 4.6 System SQ in figure 4.5 models a single-server single-buffer queueing system. The buffer has length LEN. The customers in this queueing system are people who have names enumerated in set PP. The buffer is modelled by variable sq, a sequence of people having one of the names $STEVE$, $STEVEN$, or $ESTEVE$. Initially the buffer is empty. Arrivals of new customers occur at a rate of $\frac{1}{4}$ per unit of time. The probability that a $STEVE$, a $STEVEN$, or an $ESTEVE$, arrives is $\frac{1}{3}$ for each case. The arrival process is modelled by program *arrive*. Customers are served at a rate of $\frac{1}{2}$ per unit of time. This is modelled by program *depart*. Customers are turned away if the buffer is full, i.e. $\mathsf{size}.sq = LEN$. Obviously no one is served if the buffer is empty. The service discipline of the queueing system is "First Come First Served"; customers receive service in the order of their arrival:

$sq := sq \leftarrow pp$ (arrival of a person with name pp)

$sq := \mathsf{tail}.sq$ (departure of the person at the front)

The cost structure of system SQ is set up by its sole action, *cycle*. The cost $\mid \$(\mathsf{size}.sq) \mid$ incurred during each transition is the length of the buffer at the

system SQ
constants LEN;
constraints $LEN \geq 1$;
sets
  $PP \;=\; STEVE \mid STEVEN \mid ESTEVE$;
variables
  $sq \;:\; \mathsf{seq}[LEN](PP)$;
initialisation
  $sq := \langle\,\rangle$;
programs
  $arrive \;=$
    $\mid \mathsf{size}.sq < LEN \mid; ((\bigoplus pp : PP \mid \frac{1}{3} \bullet sq := sq \leftarrow pp) \;{}_{\frac{1}{4}}\!\oplus\; \mathsf{skip})$
    $\sqcup$
    $\mid \mathsf{size}.sq = LEN \mid$;
  $depart \;=$
    $\mid \mathsf{size}.sq > 0 \mid; (sq := \mathsf{tail}.sq \;{}_{\frac{1}{2}}\!\oplus\; \mathsf{skip})$
    $\sqcup$
    $\mid \mathsf{size}.sq = 0 \mid$;
actions
  $cycle \;=\; \mid \$(\mathsf{size}.sq) \mid; (arrive; depart \sqcup depart; arrive)$;
end

Figure 4.5: Queue represented by sequence

beginning of the corresponding transition period. Arrivals and departures to and from the system take place in any order:

$$arrive; depart \sqcup depart; arrive \;.$$

We intend to show that system SQ is refined by system NQ pictured in figure 4.6.

System NQ also models a single-server single-buffer queueing system. However the buffer is represented as a natural number nq that counts the number of customers waiting in the buffer. An arrival increments nq by one and a departure decrements nq by one:

$$nq := nq + 1 \qquad \text{(arrival of a customer)}$$

$$nq := nq - 1 \qquad \text{(departure of a customer)}$$

Observe, that system NQ makes no assumption about the applied queueing discipline.

To prove $SQ \sqsubseteq NQ$ we have to find a suitable probabilistic simulation $sim : \mathcal{D}(\Gamma_{SQ}, \Gamma_{NQ})$. Short inspection of SQ and NQ suggests

$$sim \;=\; nq := \mathsf{size}.sq \;.$$

We have to prove that conditions (PS1) to (PS3) are satisfied.

$$
\begin{array}{l}
\mathsf{system}\ NQ \\
\mathsf{constants}\ LEN; \\
\mathsf{constraints}\ LEN \geq 1; \\
\mathsf{variables} \\
\quad nq \ : \ 0 \,..\, LEN; \\
\mathsf{initialisation} \\
\quad nq := 0; \\
\mathsf{programs} \\
\quad arrive \ = \\
\qquad |\ nq < LEN\ |; (nq := nq + 1 \ {}_{\frac{1}{4}}\!\oplus\ \mathsf{skip}) \\
\qquad \sqcup \\
\qquad |\ nq = LEN\ |; \\
\quad depart \ = \\
\qquad |\ nq > 0\ |; (nq := nq - 1 \ {}_{\frac{1}{2}}\!\oplus\ \mathsf{skip}) \\
\qquad \sqcup \\
\qquad |\ nq = 0\ |; \\
\mathsf{actions} \\
\quad cycle \ = \ |\ \$(nq)\ |; (arrive; depart \sqcup depart; arrive); \\
\mathsf{end}
\end{array}
$$

Figure 4.6: Queue represented by number

(PS1) The initialisations are equivalent:

$$
\begin{array}{l}
sq := \langle\,\rangle; nq := \mathsf{size}.sq \\
\quad = \ nq := \mathsf{size}.\langle\,\rangle \\
\quad = \ nq := 0\ .
\end{array}
$$

(PS2) From $\mathsf{dom}.(SQ.arrive) = \Gamma_{SQ}$ and $\mathsf{dom}.(SQ.depart) = \Gamma_{SQ}$ it follows $\mathsf{dom}.(SQ.cycle) = \Gamma_{SQ}$. We have to prove $\mathsf{dom}.(sim; NQ.cycle) = \Gamma_{SQ}$.

Analogous to the case for system SQ it holds $\mathsf{dom}.(NQ.arrive) = \Gamma_{NQ}$ and $\mathsf{dom}.(NQ.depart) = \Gamma_{NQ}$. Hence, $\mathsf{dom}.(NQ.cycle) = \Gamma_{NQ}$, and finally,

$$\mathsf{dom}.(sim; NQ.cycle) \ = \ \Gamma_{SQ}$$

because $sim \in \mathcal{D}(\Gamma_{SQ}, \Gamma_{NQ})$.

Claim (PS2) follows already from $\mathsf{dom}.sim = \Gamma_{SQ}$, and the equality of the programs $sim; NQ.cycle = SQ.cycle; sim$ proven under (PS3). We present this proof of (PS2) to demonstrate the proper use of theorem 4.4.

(PS3) We prove the stronger claim $sim; NQ.cycle = SQ.cycle; sim$. In this proof we rely heavily on the algebraic laws presented in chapter 3.

Remember that the type of the program $sim; NQ.cycle$ is $\mathcal{E}(\Gamma_{SQ}, \Gamma_{NQ})$. Types of programs are not mentioned in the proof. Let $sq \in \mathsf{seq}[LEN](PP)$

be a sequence of persons.

$$
\begin{array}{lll}
& sim; NQ.cycle & \\
= & nq := \mathsf{size}.sq; \mid \$(nq) \mid; & \\
& (NQ.arrive; NQ.depart \sqcup NQ.depart; NQ.arrive) & \\
= & \mid \$(\mathsf{size}.sq) \mid; nq := \mathsf{size}.sq; & (L29) \\
& (NQ.arrive; NQ.depart \sqcup NQ.depart; NQ.arrive) & \\
= & \mid \$(\mathsf{size}.sq) \mid; & (Q1) \\
& (SQ.arrive; SQ.depart \sqcup SQ.depart; SQ.arrive); nq := \mathsf{size}.sq & \\
= & SQ.cycle; sim &
\end{array}
$$

$$
\begin{array}{lll}
& nq := \mathsf{size}.sq; (NQ.arrive; NQ.depart \sqcup NQ.depart; NQ.arrive) & \text{(Q1)} \\
= & nq := \mathsf{size}.sq; NQ.arrive; NQ.depart \sqcup & (L9) \\
& nq := \mathsf{size}.sq; NQ.depart; NQ.arrive & \\
= & SQ.arrive; nq := \mathsf{size}.sq; NQ.depart \sqcup & (Q2) \\
& nq := \mathsf{size}.sq; NQ.depart; NQ.arrive & \\
= & SQ.arrive; SQ.depart; nq := \mathsf{size}.sq \sqcup & (Q5) \\
& nq := \mathsf{size}.sq; NQ.depart; NQ.arrive & \\
= & (SQ.arrive; SQ.depart \sqcup SQ.depart; SQ.arrive); & (Q2,Q5) \\
& nq := \mathsf{size}.sq &
\end{array}
$$

$$
\begin{array}{lll}
& nq := \mathsf{size}.sq; NQ.arrive & \text{(Q2)} \\
= & nq := \mathsf{size}.sq; \mid nq < LEN \mid; (nq := nq + 1 \;{}_{\frac{1}{4}}{\oplus}\; \mathsf{skip}) \sqcup & (L9) \\
& nq := \mathsf{size}.sq; \mid nq = LEN \mid & \\
= & \mid \mathsf{size}.sq < LEN \mid; nq := \mathsf{size}.sq; (nq := nq + 1 \;{}_{\frac{1}{4}}{\oplus}\; \mathsf{skip}) \sqcup & (L28) \\
& \mid \mathsf{size}.sq = LEN \mid; nq := \mathsf{size}.sq & \\
= & SQ.arrive; nq := \mathsf{size}.sq & (Q3)
\end{array}
$$

$$
\begin{array}{lll}
& nq := \mathsf{size}.sq; (nq := nq + 1 \;{}_{\frac{1}{4}}{\oplus}\; \mathsf{skip}) & \text{(Q3)} \\
= & (nq := \mathsf{size}.sq; nq := nq + 1) \;{}_{\frac{1}{4}}{\oplus}\; nq := \mathsf{size}.sq & (L16,L1) \\
= & ((\bigoplus pp : PP \mid \frac{1}{3} \bullet sq := sq \leftarrow pp) \;{}_{\frac{1}{4}}{\oplus}\; \mathsf{skip}); & (Q4,L17) \\
& nq := \mathsf{size}.sq &
\end{array}
$$

In the following equation remember that the type of $nq := \mathsf{size}.(sq \leftarrow pp)$ is $\mathcal{E}(\Gamma_{SQ}, \Gamma_{NQ})$. This means that the value of variable sq is not visible after its execution. So program $nq := \mathsf{size}.(sq \leftarrow pp)$ equals the program

$nq := \mathsf{size}.(sq \leftarrow pp); sq := sq \leftarrow pp$ with a tailing assignment to sq.

$$
\begin{array}{lll}
& nq := \mathsf{size}.sq; nq := nq + 1 & \text{(Q4)} \\
= & nq := \mathsf{size}.sq + 1 & (\mathit{L26}) \\
= & \bigoplus pp : PP \mid \frac{1}{3} \bullet nq := \mathsf{size}.sq + 1 & (\mathit{L14}) \\
= & \bigoplus pp : PP \mid \frac{1}{3} \bullet nq := \mathsf{size}.(sq \leftarrow pp) & \\
= & \bigoplus pp : PP \mid \frac{1}{3} \bullet (nq := \mathsf{size}.(sq \leftarrow pp); sq := sq \leftarrow pp) & \\
= & \bigoplus pp : PP \mid \frac{1}{3} \bullet (sq := sq \leftarrow pp; nq := \mathsf{size}.sq) & (\mathit{L27}) \\
= & (\bigoplus pp : PP \mid \frac{1}{3} \bullet sq := sq \leftarrow pp); nq := \mathsf{size}.sq & (\mathit{L17})
\end{array}
$$

$$
\begin{array}{lll}
& nq := \mathsf{size}.sq; NQ.depart & \text{(Q5)} \\
= & nq := \mathsf{size}.sq; \mid nq > 0 \mid; (nq := nq - 1 \ {}_{\frac{1}{2}}\!\oplus\ \mathsf{skip}) \sqcup & (\mathit{L9}) \\
& nq := \mathsf{size}.sq; \mid nq = 0 \mid & \\
= & \mid \mathsf{size}.sq > 0 \mid; nq := \mathsf{size}.sq; (nq := nq - 1 \ {}_{\frac{1}{2}}\!\oplus\ \mathsf{skip}) \sqcup & (\mathit{L28}) \\
& \mid \mathsf{size}.sq = 0 \mid; nq := \mathsf{size}.sq & \\
= & \mid \mathsf{size}.sq > 0 \mid; (sq := \mathsf{tail}.sq \ {}_{\frac{1}{2}}\!\oplus\ \mathsf{skip}); nq := \mathsf{size}.sq \sqcup & (\mathit{Q6}) \\
& \mid \mathsf{size}.sq = 0 \mid; nq := \mathsf{size}.sq & \\
= & SQ.depart; nq := \mathsf{size}.sq & (\mathit{L11})
\end{array}
$$

$$
\begin{array}{lll}
& nq := \mathsf{size}.sq; (nq := nq - 1 \ {}_{\frac{1}{2}}\!\oplus\ \mathsf{skip}) & \text{(Q6)} \\
= & (nq := \mathsf{size}.sq; nq := nq - 1) \ {}_{\frac{1}{2}}\!\oplus\ nq := \mathsf{size}.sq & (\mathit{L16,L1}) \\
= & (sq := \mathsf{tail}.sq; nq := \mathsf{size}.sq) \ {}_{\frac{1}{2}}\!\oplus\ nq := \mathsf{size}.sq & (\mathit{Q7}) \\
= & (nq := \mathsf{size}.sq \ {}_{\frac{1}{2}}\!\oplus\ \mathsf{skip}); nq := \mathsf{size}.sq & (\mathit{L1,L17})
\end{array}
$$

$$
\begin{array}{lll}
& nq := \mathsf{size}.sq; nq := nq - 1 & \text{(Q7)} \\
= & nq := \mathsf{size}.sq - 1 & (\mathit{L26}) \\
= & nq := \mathsf{size}.(\mathsf{tail}.sq) & \\
= & sq := \mathsf{tail}.sq; nq := \mathsf{size}.sq & (\mathit{L27})
\end{array}
$$

4.4.2 Equivalence

Refinement of probabilistic action systems is proven by simulation. In a refinement parts of the behaviour of the abstract system might get lost. Sometimes it is desirable to keep the entire cost behaviour intact. Equivalence proves especially useful as a tool for the analytical solution of associated performance measures (see chapter 5). A major problem in computing these measures is the size of the state space of typical systems. To make the

analytical solution feasible, the original system is replaced with an equivalent system of a much smaller size. In principle we could apply theorem 4.4 twice to prove equivalence of two systems. Instead we prefer to use the stronger version of theorem 4.7 to prove equivalence directly. A particularly simple form of simulation is called a *fusion*. It is a deterministic program that identifies states which are behaviourally indistinguishable. This technique is called *aggregation* in [58]. The approach to aggregation taken in [20, 58] is based on bisimulation between the used stochastic process algebras. It is referred to as "lumpability". In their approach the way states are "lumped" together is fixed by the definition of a bisimulation between processes. Since our approach allows for the specification of more general performance measures we need more freedom when "lumping" states. The concept of lumpability originates in the theory of Markov chains [70].

Theorem 4.7 treats simulation between equivalent probabilistic action systems. It strengthens theorem 4.4.

Theorem 4.7 Let $\mathbf{A} = (\Gamma_A, I, P)$ and $\mathbf{C} = (\Gamma_C, J, Q)$ be two probabilistic action systems. If there is a probabilistic state function $M \in \mathcal{M}(\Gamma_A, \Gamma_C)$ such that

$$J = I; M \qquad \text{(EQ1)}$$

$$M; Q = P; M \qquad \text{(EQ2)}$$

then $\mathbf{A} \equiv \mathbf{C}$.

Conditions (EQ1) and (EQ2) correspond to conditions (PS1) and (PS3) respectively. Observe that the probabilistic simulation M has domain Γ_A, not a subset as in theorem 4.4.

Example 4.8 (Example 4.6 continued) In example 4.6 we have shown $sim \in \mathcal{M}(\Gamma_{SQ}, \Gamma_{NQ})$, and

$$NQ.\mathsf{initialisation} = SQ.\mathsf{initialisation}; sim \ ,$$

$$sim; NQ.cycle = SQ.cycle; sim \ .$$

Hence probabilistic action systems SQ and NQ are equivalent. We conclude that the cost structure is independent of the queueing discipline used in system SQ.

In cases where the main objective in an application of theorem 4.7 is a reduction of the size of the state space we can replace probabilistic state function M by a state function ϕ. The intention is that state function ϕ maps states belonging to the same behaviourally equivalent class of states onto a unique representative for that class. Formally this means

$$(\phi; \phi).\tau = \phi.\tau$$

for all $\tau \in \Gamma$. A state function $\phi \in \mathcal{F}(\Gamma)$ is called *idempotent* if $\phi; \phi = \phi$.

Theorem 4.9 Let $\mathbf{A} = (\Gamma_A, I, P)$ be a probabilistic action system, and let $\phi \in \mathcal{F}(\Gamma)$ be idempotent. If

$$P; \phi \;=\; \phi; P; \phi \tag{FUS}$$

then $\mathbf{A} \equiv (\Gamma_A, I; \phi, P; \phi)$.

Condition (FUS) corresponds to conditions (PS3) and (EQ2) in theorems 4.4 and 4.7. However ϕ also expresses how the state space of the abstract system is reduced. In fact ϕ is a reduction instruction and a simulation in one. Sometimes it is easier to prove

$$P; \phi \;=\; \phi; P \tag{4.1}$$

than to prove (FUS) itself. Assuming ϕ is idempotent property (4.1) implies (FUS) because $P; \phi; \phi = P; \phi$. We call the state function ϕ in theorem 4.9 a *fusion* of system $\mathbf{A}$.

We state some results that are useful in practical applications of fusions. They concern sequential composition of fusions and sequential decomposition of the abstract action. Several fusions can be combined into a single fusion if they do not affect each other. Proposition 4.10 states this.

Proposition 4.10 Let $\mathbf{A} = (\Gamma, I, P)$ be a probabilistic action system, and let $\phi_1, \phi_2 \in \mathcal{F}(\Gamma)$ be fusions of $\mathbf{A}$. If $\phi_1; \phi_2 = \phi_2; \phi_1$, then $\phi_1; \phi_2$ is a fusion of $\mathbf{A}$.

If the abstract action P is the sequential composition $P_1; P_2$ and (FUS) holds for P_1 and P_2, then (FUS) holds also for P. This property is useful in the decomposition proofs.

Proposition 4.11 Let $P, \phi \in \mathcal{E}(\Gamma)$, $P = P_1; P_2$. If $P_1; \phi = \phi; P_1; \phi$ and $P_2; \phi = \phi; P_2; \phi$, then also $P; \phi = \phi; P; \phi$.

Note that ϕ is an extended probabilistic state relation in proposition 4.11. We use it only with fusions though.

Finally we remark that fusion does not only result in a state space reduction. It also identifies states in which an optimal system behaves similarly. This could be part of a requirement: A system has to behave in the same way in states that appear indistinguishable from an observers point of view. It is difficult to specify this requirement formally though, especially, because there is usually more than one way to identify similar states. In practice however, some ways appear more natural than others. There can also be considerable differences in the ratio of the achieved state space reduction.

4.5 Example: Polling System

In section 4.3 a model of a cyclic polling system was introduced. In the model a server walks from station to station serving packets. In this section we present an alternative model where the stations walk and the server remains at a fixed location. We show that both systems are equivalent, and compare their dimensions.

Figure 4.7 pictures the system with the fixed server. It always resides at location 1. Consequently no variable is needed to record its position. Arrivals of new packets take place at all locations that have space available in their buffer. Departures only occur from station 1, the location of the server. Program *rotate* models the walking stations. Each station is replaced with its successor within the cyclic arrangement.

Variable *station* in system *POLLING* represents the location of the server. In system *REDPOLL* the location of the server is 1 and does not change. So, relative to the location of the server in both systems, station 1 in system *REDPOLL* corresponds to the station identified by variable *station* in system *POLLING*. In general, station s in system *REDPOLL* corresponds to station $\langle\!\langle s + station \rangle\!\rangle$, where

$$\langle\!\langle x \rangle\!\rangle \;\; \widehat{=} \;\; ((x - 2) \;\mathsf{mod}\; \mathit{STATIONS}) + 1 \;.$$

We note that the function

$$(\lambda\, s : \mathit{STATION} \bullet \langle\!\langle s + \mathit{station} \rangle\!\rangle) \tag{P0}$$

is bijective for a fixed $\mathit{station} \in \mathit{STATION}$. It relocates stations from its range to corresponding stations in its domain. Based on the function $\langle\!\langle \cdot \rangle\!\rangle$ we define a probabilistic simulation $\mathit{sim} : \mathcal{M}(\Gamma_{\mathit{POLLING}}, \Gamma_{\mathit{REDPOLL}})$ by

$$\begin{aligned} \mathit{sim} \;\; = \;\; & \mathit{mov} := \mathit{moving} \;\| \\ & \mathit{loc} := (\lambda\, s : \mathit{STATION} \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle) \;. \end{aligned}$$

Program *sim* copies the contents buffered at the stations of system *POLLING* to the corresponding stations of system *REDPOLL*.

We have to prove that *sim* satisfies (EQ1) and (EQ2) in theorem 4.7. We present the proof in detail because we think the algebraic reasoning used is an important property of the approach. The proven result itself is perhaps less interesting.

(EQ1) The initialisation of system *POLLING* can simulate the initialisation of system *REDPOLL*. Let $ST = \mathit{STATION}$ in:

$$\begin{aligned} & \mathit{POLLING}.\mathsf{initialisation}; \mathit{sim} \\ & \quad = \;\; \mathit{station}, \mathit{moving}, \mathit{buffer} := 1, \mathsf{false}, ST \times \{0\}; \end{aligned}$$

system $REDPOLL$
constants
  $STATIONS$; $CAPACITY$;
constraints
  $STATIONS > 0 \wedge CAPACITY > 0$;
sets
  $STATION \;=\; 1 \mathrel{..} STATIONS$;
  $NEXT \;=\; (\lambda\, s : 1 \mathrel{..} (STATIONS - 1) \bullet s + 1) \cup \{STATIONS \mapsto 1\}$;
  $ARRIVAL \;=\; \{\mathsf{true} \mapsto 0.1, \mathsf{false} \mapsto 0.9\}$;
variables
  $loc \;:\; STATION \rightarrow 0 \mathrel{..} CAPACITY$;
  $mov \;:\; \mathbb{B}$;
programs
  $arrival$ = // arrival of packets at locations
    $\bigoplus S : \mathbb{P}(loc^{\sim}[0 \mathrel{..} CAPACITY - 1])$
      $\mid (\prod s : loc^{\sim}[0 \mathrel{..} CAPACITY - 1] \bullet ARRIVAL.(s \in S)) \bullet$
        $loc := loc \lhd\!\!+ (\lambda\, s : S \bullet loc.s + 1)$;
  $departure$ = // departure of processed packets from location 1
    $\mid loc.1 > 0 \mid$;
      $(loc := loc \lhd\!\!+ \{1 \mapsto loc.1 - 1\}\ {}_{0.25}\!\oplus\ \mathsf{skip})$
    $\sqcup$
    $\mid loc.1 = 0 \mid$;
  $rotate$ = // rotate locations
    $loc := (\lambda\, s : STATION \bullet loc.(NEXT.s))$;
initialisation
  $mov := \mathsf{false} \parallel loc := STATION \times \{0\}$;
actions
  $serve$ =
    $\mid \sum s : STATION \bullet loc.s \mid$;
    $\mid \neg\, mov \mid$; $arrival$; $departure$;
  $walk$ =
    $\mid 1.0 \mid$;
    $\mid \sum s : STATION \bullet loc.s \mid$;
    $arrival$;
    $(mov := \mathsf{true}\ {}_{0.5}\!\oplus\ (mov := \mathsf{false} \parallel rotate))$;
end

Figure 4.7: Another cyclic polling system

$$\begin{aligned}
& mov, loc := moving, (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle) \\
= \;& mov, loc := \mathsf{false}, (\lambda\, s : ST \bullet (ST \times \{0\}).\langle\!\langle s + 1 \rangle\!\rangle) \\
= \;& mov, loc := \mathsf{false}, (\lambda\, s : ST \bullet (ST \times \{0\}).s) \\
= \;& mov, loc := \mathsf{false}, ST \times \{0\} \\
= \;& REDPOLL.\mathsf{initialisation}\ .
\end{aligned}$$

(EQ2) We have to prove that the action of system *POLLING* can simulate the action of system *REDPOLL*, but does not allow additional behaviour. We use the same modular style of proof as in example 4.6:

$$\begin{aligned}
& sim; (REDPOLL.serve \sqcup REDPOLL.walk) \\
= \;& sim; REDPOLL.serve \sqcup sim; REDPOLL.walk && (L9) \\
= \;& POLLING.serve; sim \sqcup POLLING.walk; sim && (P1,P7) \\
= \;& (POLLING.serve \sqcup POLLING.walk); sim && (L11)
\end{aligned}$$

Simulation of action *serve*: In the following we let $ST = STATION$. We also let $CC = 0\,..\,CAPACITY - 1$. We also let $ST = STATION$, and $A = ARRIVAL$.

$$\begin{aligned}
& sim; REDPOLL.serve && \text{(P1)} \\
= \;& sim; |\textstyle\sum s : ST \bullet loc.s\,|; \\
& |\neg\, mov\,|; REDPOLL.arrival; REDPOLL.departure \\
= \;& |\textstyle\sum s : ST \bullet buffer.s\,|; sim; && (P2) \\
& |\neg\, mov\,|; REDPOLL.arrival; REDPOLL.departure \\
= \;& |\textstyle\sum s : ST \bullet buffer.s\,|; && (L28) \\
& |\neg\, moving\,|; sim; REDPOLL.arrival; REDPOLL.departure \\
= \;& |\textstyle\sum s : ST \bullet buffer.s\,|; && (P3) \\
& |\neg\, moving\,|; POLLING.arrival; sim; REDPOLL.departure \\
= \;& |\textstyle\sum s : ST \bullet buffer.s\,|; && (P4) \\
& |\neg\, moving\,|; POLLING.arrival; POLLING.departure; sim \\
= \;& POLLING.serve; sim
\end{aligned}$$

$$\begin{aligned}
& sim; |\textstyle\sum s : ST \bullet loc.s\,| && \text{(P2)} \\
= \;& |\textstyle\sum s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle\,|; sim && (L29) \\
= \;& |\textstyle\sum s : ST \bullet buffer.s\,|; sim\ . && (P0)
\end{aligned}$$

$$\begin{aligned}
& sim; REDPOLL.arrival && \text{(P3)} \\
= \;& mov := moving; && (L30)
\end{aligned}$$

$$
\begin{aligned}
& \quad (loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle); REDPOLL.arrival) \\
= & \quad mov := moving; && (L33) \\
& \quad \bigoplus S : \mathbb{P}((\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle)^{\sim}[CC]) \\
& \qquad \mid (\prod s : (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle)^{\sim}[CC] \bullet A.(s \in S)) \bullet \\
& \qquad\quad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle) \\
& \qquad\qquad\quad \mathbin{\lhd\!\!\!+} (\lambda\, s : S \bullet (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle).s + 1) \\
= & \quad mov := moving; && (P0) \\
& \quad \bigoplus S : \mathbb{P}(buffer^{\sim}[CC]) \\
& \qquad\quad \mid (\prod s : buffer^{\sim}[CC] \bullet A.(s \in S)) \bullet \\
& \qquad\qquad loc := \\
& \qquad\qquad\quad (\lambda\, r : ST \bullet \\
& \qquad\qquad\qquad (buffer \mathbin{\lhd\!\!\!+} (\lambda\, s : S \bullet buffer.s + 1)).\langle\!\langle r + station \rangle\!\rangle) \\
= & \quad mov := moving; && (L27) \\
& \quad \bigoplus S : \mathbb{P}(buffer^{\sim}[CC]) \\
& \qquad\quad \mid (\prod s : buffer^{\sim}[CC] \bullet A.(s \in S)) \bullet \\
& \qquad\qquad buffer := buffer \mathbin{\lhd\!\!\!+} (\lambda\, s : S \bullet buffer.s + 1); \\
& \qquad\qquad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle) \\
= & \quad \bigoplus S : \mathbb{P}(buffer^{\sim}[CC]) && (L33, L27, L30) \\
& \qquad\quad \mid (\prod s : buffer^{\sim}[CC] \bullet A.(s \in S)) \bullet \\
& \qquad\qquad buffer := buffer \mathbin{\lhd\!\!\!+} (\lambda\, s : S \bullet buffer.s + 1); sim \\
= & \quad POLLING.arrival; sim && (L17)
\end{aligned}
$$

$$
\begin{aligned}
& sim; REDPOLL.departure && \text{(P4)} \\
= & \quad sim; \mid loc.1 > 0 \mid; && (L9) \\
& \qquad (loc := loc \mathbin{\lhd\!\!\!+} \{1 \mapsto loc.1 - 1\} \;_{0.25}\!\oplus \mathsf{skip}) \sqcup \\
& \quad sim; \mid loc.1 = 0 \mid \\
= & \quad \mid buffer.station > 0 \mid; sim; && (L28, P5) \\
& \qquad (loc := loc \mathbin{\lhd\!\!\!+} \{1 \mapsto loc.1 - 1\} \;_{0.25}\!\oplus \mathsf{skip}) \sqcup \\
& \quad \mid buffer.station = 0 \mid \\
= & \quad \mid buffer.station > 0 \mid; && (L30, L27, P6) \\
& \qquad (buffer := buffer \mathbin{\lhd\!\!\!+} \{station \mapsto buffer.station - 1\}; sim \;_{0.25}\!\oplus \\
& \qquad\quad \mathsf{skip}; sim) \sqcup \\
& \quad \mid buffer.station = 0 \mid; sim \\
= & \quad POLLING.departure; sim && (L17, L11)
\end{aligned}
$$

$$
[loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station \rangle\!\rangle)]\,(loc.1) \qquad \text{(P5)}
$$

$$\begin{aligned}
&= (\lambda\, s : ST \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle).1 \\
&= \mathit{buffer}.\mathit{station}
\end{aligned}$$

$$\begin{aligned}
& \mathit{loc} := (\lambda\, s : ST \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle); && \text{(P6)} \\
& \mathit{loc} := \mathit{loc} \oplus \{1 \mapsto \mathit{loc}.1 - 1\} \\
&\quad = \mathit{loc} := (\lambda\, s : ST \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle) && (\mathit{L26}) \\
&\qquad \oplus \{1 \mapsto (\lambda\, s : ST \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle).1 - 1\} \\
&\quad = \mathit{loc} := (\lambda\, s : ST \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle) && (\mathit{P5}) \\
&\qquad \oplus \{1 \mapsto \mathit{buffer}.\mathit{station} - 1\} \\
&\quad = \mathit{loc} := (\lambda\, s : ST \bullet \\
&\qquad (\mathit{buffer} \oplus \{\mathit{station} \mapsto \mathit{buffer}.\mathit{station} - 1\}).\langle\!\langle s + \mathit{station} \rangle\!\rangle) \\
&\quad = \mathit{buffer} := \mathit{buffer} \oplus \{\mathit{station} \mapsto \mathit{buffer}.\mathit{station} - 1\}; && (\mathit{L27}) \\
&\qquad \mathit{loc} := (\lambda\, s : ST \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle)
\end{aligned}$$

Simulation of action *walk*: Let $ST = STATION$ and $bb \in \mathbb{B}$.

$$\begin{aligned}
& \mathit{sim}; \mathit{REDPOLL}.\mathit{walk} && \text{(P7)} \\
&\quad = \mathit{sim}; |\ 1.0\ |; |\ \textstyle\sum s : ST \bullet \mathit{loc}.s\ |; \mathit{REDPOLL}.\mathit{arrival}; \\
&\qquad (\mathit{mov} := \mathsf{true}\ {}_{0.5}\!\oplus (\mathit{mov} := \mathsf{false} \parallel \mathit{rotate})) \\
&\quad = |\ 1.0\ |; \mathit{sim}; |\ \textstyle\sum s : ST \bullet \mathit{loc}.s\ |; \mathit{REDPOLL}.\mathit{arrival}; && (\mathit{L29}) \\
&\qquad (\mathit{mov} := \mathsf{true}\ {}_{0.5}\!\oplus (\mathit{mov} := \mathsf{false} \parallel \mathit{rotate})) \\
&\quad = |\ 1.0\ |; |\ \textstyle\sum s : ST \bullet \mathit{buffer}.s\ |; \mathit{sim}; \mathit{REDPOLL}.\mathit{arrival}; && (\mathit{P2}) \\
&\qquad (\mathit{mov} := \mathsf{true}\ {}_{0.5}\!\oplus (\mathit{mov} := \mathsf{false} \parallel \mathit{rotate})) \\
&\quad = |\ 1.0\ |; |\ \textstyle\sum s : ST \bullet \mathit{buffer}.s\ |; \mathit{POLLING}.\mathit{arrival}; \mathit{sim}; && (\mathit{P3}) \\
&\qquad (\mathit{mov} := \mathsf{true}\ {}_{0.5}\!\oplus (\mathit{mov} := \mathsf{false} \parallel \mathit{rotate})) \\
&\quad = |\ 1.0\ |; |\ \textstyle\sum s : ST \bullet \mathit{buffer}.s\ |; \mathit{POLLING}.\mathit{arrival}; && (\mathit{L16,L30}) \\
&\qquad (\mathit{sim}; \mathit{mov} := \mathsf{true}\ {}_{0.5}\!\oplus \mathit{sim}; \mathit{mov} := \mathsf{false}; \mathit{rotate}) \\
&\quad = |\ 1.0\ |; |\ \textstyle\sum s : ST \bullet \mathit{buffer}.s\ |; \mathit{POLLING}.\mathit{arrival}; && (\mathit{P8}) \\
&\qquad (\mathit{moving} := \mathsf{true}; \mathit{sim}\ {}_{0.5}\!\oplus \mathit{moving} := \mathsf{false}; \mathit{sim}; \mathit{rotate}) \\
&\quad = |\ 1.0\ |; |\ \textstyle\sum s : ST \bullet \mathit{buffer}.s\ |; \mathit{POLLING}.\mathit{arrival}; && (\mathit{P9}) \\
&\qquad (\mathit{moving} := \mathsf{true}; \mathit{sim}\ {}_{0.5}\!\oplus \\
&\qquad\quad \mathit{moving} := \mathsf{false}; \mathit{station} := \mathit{NEXT}.\mathit{station}; \mathit{sim}) \\
&\quad = \mathit{POLLING}.\mathit{walk}; \mathit{sim} && (\mathit{L17,L30})
\end{aligned}$$

$$\begin{aligned}
& \mathit{sim}; \mathit{mov} := bb && \text{(P8)} \\
&\quad = (\mathit{mov} := \mathit{moving}; \mathit{mov} := bb); && (\mathit{L30,L27}) \\
&\qquad \mathit{loc} := (\lambda\, s : ST \bullet \mathit{buffer}.\langle\!\langle s + \mathit{station} \rangle\!\rangle)
\end{aligned}$$

$$
\begin{aligned}
&= (mov := bb); && (L26)\\
&\quad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station\rangle\!\rangle)\\
&= (moving := bb; mov := moving); && (L27)\\
&\quad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station\rangle\!\rangle)\\
&= moving := bb; sim && (L30)
\end{aligned}
$$

$$
\begin{aligned}
∼ rotate && \text{(P9)}\\
&= mov := moving; && (L30)\\
&\quad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station\rangle\!\rangle);\\
&\quad loc := (\lambda\, s : ST \bullet loc.(NEXT.s))\\
&= mov := moving; && (L26)\\
&\quad loc := (\lambda\, s : ST \bullet (\lambda\, r : ST \bullet buffer.\langle\!\langle r + station\rangle\!\rangle).(NEXT.s))\\
&= mov := moving;\\
&\quad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle NEXT.s + station\rangle\!\rangle)\\
&= mov := moving; && (P10)\\
&\quad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + NEXT.station\rangle\!\rangle)\\
&= mov := moving; && (L27)\\
&\quad station := NEXT.station;\\
&\quad loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station\rangle\!\rangle)\\
&= station := NEXT.station; && (L27)\\
&\quad mov := moving; loc := (\lambda\, s : ST \bullet buffer.\langle\!\langle s + station\rangle\!\rangle)\\
&= station := NEXT.station; sim && (L30)
\end{aligned}
$$

To prove

$$\langle\!\langle NEXT.s + station\rangle\!\rangle = \langle\!\langle s + NEXT.station\rangle\!\rangle \qquad \text{(P10)}$$

we distinguish four cases:

$s < STATIONS \wedge station < STATIONS$:

$$
\begin{aligned}
&\langle\!\langle NEXT.s + station\rangle\!\rangle\\
&= ((s + 1 + station - 2) \;\mathsf{mod}\; STATIONS) + 1\\
&= ((s + station + 1 - 2) \;\mathsf{mod}\; STATIONS) + 1\\
&= \langle\!\langle s + NEXT.station\rangle\!\rangle
\end{aligned}
$$

$s = STATIONS \wedge station < STATIONS$:

$$
\begin{aligned}
&\langle\!\langle NEXT.s + station\rangle\!\rangle\\
&= ((1 + station - 2) \;\mathsf{mod}\; STATIONS) + 1\\
&= ((STATIONS + station + 1 - 2) \;\mathsf{mod}\; STATIONS) + 1\\
&= \langle\!\langle s + NEXT.station\rangle\!\rangle
\end{aligned}
$$

$s < STATIONS \wedge station = STATIONS$:

$$\begin{aligned} & \langle\!\langle NEXT.s + station \rangle\!\rangle \\ = \; & ((s + 1 + STATIONS - 2) \ \mathsf{mod} \ STATIONS) + 1 \\ = \; & ((s + 1 - 2) \ \mathsf{mod} \ STATIONS) + 1 \\ = \; & \langle\!\langle s + NEXT.station \rangle\!\rangle \end{aligned}$$

$s = STATIONS \wedge station = STATIONS$:

$$\begin{aligned} & \langle\!\langle NEXT.s + station \rangle\!\rangle \\ = \; & ((1 + STATIONS - 2) \ \mathsf{mod} \ STATIONS) + 1 \\ = \; & ((STATIONS + 1 - 2) \ \mathsf{mod} \ STATIONS) + 1 \\ = \; & \langle\!\langle s + NEXT.station \rangle\!\rangle \end{aligned}$$

Observations In this example it seemed appropriate to change the state space by removing variable *station* from system *POLLING*. It is not needed because its value would always be 1. We could have used a fusion but found the equivalence proof easier. See chapter 7 for an example of a fusion being applied to a larger system.

We can easily calculate the sizes of the state spaces $\Gamma_{POLLING}$ and $\Gamma_{REDPOLL}$:

$$\begin{aligned} & \mathsf{card}.\Gamma_{POLLING} \\ = \; & \mathsf{card}.STATION \\ & * (\mathsf{card}.(0 \mathinner{..} CAPACITY))^{\mathsf{card}.STATION} \\ & * \mathsf{card}.\mathbb{B} \\ = \; & STATIONS * (CAPACITY + 1)^{STATIONS} * 2 \end{aligned}$$

$$\begin{aligned} & \mathsf{card}.\Gamma_{REDPOLL} \\ = \; & (\mathsf{card}.(0 \mathinner{..} CAPACITY))^{\mathsf{card}.STATION} * \mathsf{card}.\mathbb{B} \\ = \; & (CAPACITY + 1)^{STATIONS} * 2 \end{aligned}$$

It is not difficult to convince oneself that $\mathsf{reach}.\mathbf{POLLING}$ equals the whole state space $\Gamma_{POLLING}$, and $\mathsf{reach}.\mathbf{REDPOLL}$ equals $\Gamma_{REDPOLL}$. Obviously the simple relationship

$$\mathsf{card}.(\mathsf{reach}.\mathbf{POLLING}) \;=\; STATIONS * \mathsf{card}.(\mathsf{reach}.\mathbf{REDPOLL})$$

holds between the two systems. In other words, the state space of system *REDPOLL* is linearly smaller in the number of stations than that of system *POLLING*.

We also observe that, if *moving*, or *mov*, is true only action *walk* may occur; and if *moving*, or *mov*, is false both actions may occur. Because both actions are deterministic the size of $\mathsf{card}.(serve \sqcup walk)$ can be calculated as $\frac{3}{2} * \mathsf{card}.\Gamma$, where Γ is the state space of the corresponding system.

4.6 Remarks

Probabilistic action systems are closely related to probabilistic automata [99], probabilistic and nondeterministic systems [17], probabilistic concurrent programs [42], and simple probabilistic automata [107]. Probabilistic automata are a probabilistic extension of classical automata theory. Using a similar model [40] propose a method of model checking real-time temporal properties of probabilistic automata. The temporal properties then can be checked for specified probabilistic bounds on the timing behaviour of an automaton. The underlying model of time is similar to ours in that from one transition to the next one unit of time is consumed. Probabilistic and nondeterministic systems are used to check probabilistic temporal logic formulas over Markov decision processes. Similar to our formalism, nondeterminism and probability in [17] are separate concepts. It corresponds closely to the construction time nondeterminism of [48], and similarly [77]. Following [17] the article [25] treats model checking of timing and reliability properties by estimating bounds on the quantities of interest. The computational aspect of the approach is based on Markov decision processes as well. In [17] it is demonstrated how this can be done by solving a linear programming problem. Probabilistic concurrent programs are used to analyse the scheduling of a number of processes, each of which is described by a probability matrix. Simple probabilistic automata extend labelled transition systems. In [107] simulations and bisimulations for simple probabilistic automata are defined, and it is shown that certain temporal properties are preserved by them. However, the behaviour of the named formalisms is described by probabilistic state transitions. Properties of specifications consequently concern their probabilistic behaviour. General performance measures based on expected costs are not representable. An approach similar to [107] is taken in [66] where temporal logic formulas are refined into deterministic probabilistic transition systems. The temporal logic specifications are inherently nondeterministic (and determine a collection deterministic implementations). The author [121] again follows an approach similar to [66] but based on the process algebra CCS [83]. A notion of probabilistic simulation is defined and algebraic laws are presented. In [65] it is suggested to model the behaviour of probabilistic and nondeterministic processes based on reward testing instead of probabilistic testing. This is the origin of the idea to describe the behaviour of probabilistic action systems by traces of expected costs. Simulation between processes is not treated in [65] though.

The above formalisms can be used to derive quantitative properties of a system. Some formalism use similar semantical models but serve to reason about qualitative properties. It is proven that some property holds with probability 1 or 0. In all these formalisms a close correspondence between probability and fairness is shown. We discuss some of them briefly in the remainder of this paragraph. Concurrent Markov chains [117] are similar to

Markov decision processes but distinguish between probabilistic and nondeterministic states. The article [117] deals with the automatic verification of standard linear temporal logic properties of concurrent Markov chains. Similarly, in [97] temporal properties of concurrent, but deterministic systems, are analysed. It is extended in [96] to include nondeterminism. However instead of choosing one successor state probabilistically, a set of successor states is chosen adding extra nondeterminism. This extension is not covered by Markov decision processes. In [100] probability is added to the UNITY formalism. It is used for qualitative analysis of temporal properties. The article also contains a host of proof rules.

In the article [82] the authors investigate probabilistic data refinement of probabilistic predicate transformers. The simulation-based proof rules presented are similar to the ones for standard data refinement [35, 36]. This is in correspondence with our work. The probabilistic simulation of theorem 4.4 was developed using the machine simulation of proposition 2.1 as a blue print. Unlike for machines however we have not established completeness. The requirement that probabilistic simulations must be deterministic suggests that we are quite far away from such a result. Completeness has yet to be investigated. We have also not yet considered unbounded nondeterminism. A problem with infinite state systems in general is that they may not have optimal implementations. Optimality of probabilistic action systems is the subject of the next chapter.

Chapter 5

Optimal Systems

Markov decision processes are an alternative way to model live systems. They are associated with optimality criteria that express performance objectives to be met. There is a well-established theory [28, 98, 110, 115] containing both general existence results of optimal solutions to the posed optimisation problem, and algorithms to compute them.

The behaviour of Markov decision processes and probabilistic action systems is closely related. We introduce average cost optimality of probabilistic action systems by exploiting this relationship. We consider average cost optimality appropriate for the systems analysed in this thesis. There are other criteria that may be more appropriate in other applications. Two of them are briefly discussed in section 5.2.

5.1 Markov Decision Processes

Markov decision processes are a classical model used in performance analysis [16, 28, 61, 62, 98, 110, 115]. We briefly review the discrete-time model presented in [115]. Discrete-time means that processes are observed at equidistant points of time $0, 1, 2, \ldots$ (see chapter 4). At time n a process M is in some state $\tau \in \Gamma$ and a decision has to be made on the next action to take. The set of possible states is denoted by Γ. For each state τ there is a set of possible actions $A(\tau)$. Both the state space Γ and the sets $A(\tau)$, $\tau \in \Gamma$, are assumed to be finite. If process M chooses action $a \in A(\tau)$ in state τ, then it incurs a cost $c_\tau(a)$, and at time $n+1$ it will be in state τ' with probability $p_{\tau\tau'}(a)$ where $\sum_{\tau'} p_{\tau\tau'}(a) = 1$. The incurred cost $c_\tau(a)$ and transition probabilities $p_{\tau\tau'}(a)$ are independent of the history of process M.

A Markov decision process may be modelled in our notation by a tuple $\mathbf{M} = (\Gamma, P)$ where

> Γ is a state space, and
>
> $P \in \mathcal{E}(\Gamma)$, with $\mathsf{dom}.P = \Gamma$, is a program.

The condition $\mathsf{dom}.P = \Gamma$ ensures that the process cannot deadlock. In other words only systems that are in ongoing operation are modelled. If $\mathbf{A} = (\Gamma, I, P)$ is a live probabilistic action system, then $\mathsf{reach}.\mathbf{A}$ is a subset of $\mathsf{dom}.P$. Hence live system $\mathbf{A}$ corresponds to a Markov decision process. We define the Markov decision process $\mathsf{proc}.\mathbf{A}$ associated with a live system $\mathbf{A}$ by

$$\mathsf{proc}.\mathbf{A} \;\widehat{=}\; (\mathsf{reach}.\mathbf{A}, (\mathsf{reach}.\mathbf{A}) \lhd P) \ .$$

The behaviour of Markov decision processes is determined by policies [115]. A policy is a prescription for taking actions at each stage in the evolution of a process. In our notation a *finite policy* is modelled by a pair consisting of a sequence of cost functions and a sequence of probabilistic state functions:

$$\begin{aligned} \mathsf{po}.\mathbf{M}.n.(\kappa, \Phi) \;\widehat{=}\;& \mathsf{size}.\kappa = n \wedge \\ & \mathsf{size}.\Phi = n \wedge \\ & \forall\, i : 1 \mathrel{..} n \bullet (\kappa.i, \Phi.i) \in \mathsf{fun}.P \ . \end{aligned}$$

The behaviour of $\mathbf{M}$ is defined as the collection of its finite policies. Remember that $(\Pi.\Phi).\tau.\tau'$ denotes the conditional probability of being in state τ' after execution of sequence Φ, having started in state τ. Let $(\kappa, \Phi) \in \mathsf{po}.\mathbf{M}.n$ be a finite policy of length n. Then

$$(\Pi.(\Phi \uparrow j - 1) * \kappa.j) : \Gamma \to \mathbb{R}_{\geq 0} \ , \tag{5.1}$$

for $j \in 1 \mathrel{..} n$, describes the conditional expected costs that $\mathbf{M}$ incurs at the last transition if it follows policy (κ, Φ) till time j. The sequence of cost functions in (5.1) has its correspondence in a finite trace of a probabilistic action system from which $\mathbf{M}$ stems. Proposition 5.1 establishes this correspondence between traces and policies.

Proposition 5.1 Let $\mathbf{A} = (\Gamma, I, P)$ be a live probabilistic action system. For all $t : \mathsf{seq}\,\mathbb{R}_{\geq 0}$ and $g : \mathbb{D}\,\Gamma$:

$$\begin{aligned} \mathsf{path}.\mathbf{A}.t.g \;\Leftrightarrow\; & \exists\, f : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.t) \bullet \\ & g = f * \Pi.\Phi \wedge \\ & \forall\, j : \mathsf{dom}.t \bullet t.j = f * \Pi.(\Phi \uparrow j - 1) * \kappa.j \ . \end{aligned}$$

Define the predicate $\mathsf{trace}.f.(\kappa, \Phi).t$, associating a finite policy (κ, Φ) with a finite trace t, by $(\forall\, j : \mathsf{dom}.t \bullet t.j = f * \Pi.(\Phi \uparrow j - 1) * \kappa.j)$. Using this notation, proposition 5.1 essentially says:

Corollary 5.2 For all $t : \mathsf{seq}\,\mathbb{R}_{\geq 0}$:

$$\mathsf{tr}.\mathbf{A}.t \Leftrightarrow (\exists\, f : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.t) \bullet \mathsf{trace}.f.(\kappa, \Phi).t) \ .$$

A live system can be described by its infinite traces only. Similarly, for Markov decision processes infinite policies are sufficient to describe their behaviour. The infinite policies of a Markov decision process $\mathbf{M}$ are defined by

$$\mathsf{ipo}.\mathbf{M}.(\kappa, \Phi) \;\widehat{=}\; \forall\, i : \mathbb{N}_1 \bullet (\kappa.i, \Phi.i) \in \mathsf{fun}.P \;.$$

An infinite policy (κ, Φ) is called *stationary* if $\kappa.n = \kappa.1$ and $\Phi.n = \Phi.1$ for all $n \in \mathbb{N}_1$. We sometimes denote stationary policies by (C, M) where $(C, M) \in \mathsf{fun}.P$.

Because the finite policies of $\mathsf{proc}.\mathbf{A}$ induce the finite traces of $\mathbf{A}$ we expect the same to be true for infinite policies and infinite traces. Let (κ, Φ) be an infinite policy of a Markov decision process over a state space Γ, and $f : \mathbb{D}\,\Gamma$ a density. The infinite trace associated with f and (κ, Φ) is defined by:

$$\mathsf{itrace}.f.(\kappa, \Phi).v \;\widehat{=}\; \forall\, j : \mathbb{N}_1 \bullet v.j = f * \Pi.(\Phi \uparrow j - 1) * \kappa.j \;.$$

As in proposition 5.2 the initial densities f of a probabilistic action system $\mathbf{A}$ are required to achieve an exact match between an infinite trace v and an infinite policy (κ, Φ) of the associated Markov decision process $\mathsf{proc}.\mathbf{A}$.

Theorem 5.3 Let $\mathbf{A} = (\Gamma, I, P)$ be live. For all $v \in \mathsf{seq}_\infty\, \mathbb{R}_{\geq 0}$:

$$\mathsf{itr}.\mathbf{A}.v \;\Leftrightarrow\; \exists\, f : I,\, (\kappa, \Phi) : \mathsf{ipo}.(\mathsf{proc}.\mathbf{A}) \bullet \mathsf{itrace}.f.(\kappa, \Phi).v \;.$$

This correspondence between $\mathbf{A}$ and $\mathsf{proc}.\mathbf{A}$ is used in section 5.3 to carry over the concept of optimality of Markov decision processes, defined in section 5.2, to probabilistic action systems.

5.2 Optimisation Criteria

We introduce three optimisation criteria that are commonly used. More criteria can be found in the literature [68, 98, 110]. We do not treat all of them because all essential ideas are already present in the three criteria. In this study we make use of the average-cost criterion. We think that knowledge of the other two, the finite-horizon criterion and the discounted-cost criterion, helps to understand the nature of average-cost optimality. All introduced entities exist for systems with finite state spaces [98]. Note that probabilistic action systems allow the values of the parameters in the constants section to be left open. An optimal system, however, can only be determined for particular choices of parameters. As a consequence an optimal system for some choice of parameter may not be optimal for a different choice. In the following let $\mathbf{M} = (\Gamma, P)$ be a Markov decision process.

The finite-horizon criterion considers only a fixed number n of transitions. Let (κ, Φ) be a finite policy. Then $\mathsf{tot}.(\kappa, \Phi).n.\tau \in \mathbb{R}_{\geq 0}$ is the total expected cost that $\mathbf{M}$ would incur after n transitions having started in state $\tau \in \Gamma$,

$$\mathsf{tot}.(\kappa, \Phi).n \;\; \widehat{=} \;\; \sum_{m=1}^{n} \Pi.(\Phi \uparrow m-1) * \kappa.m \; .$$

The optimal value $\mathsf{fin}.\mathbf{M}.n$ of Markov decision process $\mathbf{M}$ with respect to the finite-horizon criterion of length n is defined as the infimum of $\mathsf{tot}.(\kappa, \Phi).n$ over all policies of length n:

$$\mathsf{fin}.\mathbf{M}.n \;\; = \inf_{(\kappa,\Phi) \in \mathsf{po}.\mathbf{M}.n} \mathsf{tot}.(\kappa, \Phi).n \; . \tag{5.2}$$

Note that the infimum in (5.2) is taken component-wise. In the application of the finite-horizon criterion the choice of the length n of the horizon is crucial. If it is chosen too short a policy (κ, Φ) might be regarded as optimal where a costly transition occurs past the horizon. Yet there could be better policies. This makes it difficult to trust finite-horizon optimal policies except in cases where n is known in advance (see [98] for examples).

As n approaches infinity so might the total expected cost $\mathsf{tot}.(\kappa, \Phi).n$. There are two major ways to achieve finiteness in this situation: discounting and averaging. First we treat the concept of discounting. A discount factor $\alpha \in (0, 1)$ is used so that future costs are discounted at rate α. Costs occurring further in the future are valued less than costs occurring in the immediate future. In the definition of the discounted expected cost, $\mathsf{rab}.\alpha.(\kappa, \Phi)$, the expected cost at the m-th transition is discounted with factor α^{m-1}, and then the infinite sum is taken over all $m \in \mathbb{N}_1$. Note that the first transition is not discounted at all:

$$\mathsf{rab}.\alpha.(\kappa, \Phi) \;\; \widehat{=} \;\; \sum_{m=1}^{\infty} \alpha^{m-1} \Pi.(\Phi \uparrow m-1) * \kappa.m \; .$$

Let α be fixed. We define the optimal value of $\mathbf{M}$ with respect to the discounted-cost criterion. It is the infimum of $\mathsf{rab}.\alpha.(\kappa, \Phi)$ over all infinite policies:

$$\inf_{(\kappa,\Phi) \in \mathsf{ipo}.\mathbf{M}} \mathsf{rab}.\alpha.(\kappa, \Phi) \; . \tag{5.3}$$

The optimum attained in (5.3) overemphasises the present in comparison to the future. Let $\alpha = 0.9$, then the first transition is discounted with factor $\alpha^0 = 1$, and the tenth transition with factor $\alpha^{10} \approx 0.35$. After twenty transitions it is $\alpha^{20} \approx 0.1$. This is acceptable or required in some models. For instance, in economical models where, say, future loss is discounted to account for inflation. In other models like the polling system (see section 4.3)

discounting is unsuitable. There waiting packages ought to be valued with the same weight at all times. This is realised by the average-cost criterion.

The average cost incurred in the long-run, $\mathsf{avg}.(\kappa, \Phi)$, by following policy (κ, Φ) is defined by

$$\mathsf{avg}.(\kappa, \Phi) \;\hat{=}\; \limsup_{n\to\infty} \frac{\mathsf{tot}.(\kappa, \Phi).n}{n} \; .$$

The term $\frac{1}{n} * \mathsf{tot}.(\kappa, \Phi).n$ represents the average cost incurred over the first n transitions. We let n approach infinity to get the long-run average-cost. The limit superior is used because the limit need not exists for all policies. It exists, however, for all stationary policies [98]. The use of the limit superior corresponds to the choice of the highest cost, i.e. the worst case from that perspective. This seems appropriate when it is the aim to minimise the cost a system may possibly incur. The optimal average-cost value, $\mathsf{opt}.\mathbf{M}$, of a Markov decision process $\mathbf{M}$ is defined by:

$$\mathsf{opt}.\mathbf{M} \;\hat{=}\; \inf_{(\kappa,\Phi)\in\mathsf{ipo}.\mathbf{M}} \mathsf{avg}.(\kappa, \Phi) \; . \tag{5.4}$$

We have chosen average-cost optimality as performance measure in this work because it is appropriate for systems in continual operation [78], and simple. It is simpler than the other two because there is no additional parameter n or α required. Both parameters appear difficult to estimate if they are not already part of the model being analysed.

5.3 Average Cost Optimality

We introduce average cost optimality of live probabilistic action systems and show that it reduces to average cost optimality of the associated Markov decision process.

The average-cost optimal value of a live system $\mathbf{A}$ is defined in terms of infinite traces of costs by

$$\mathsf{val}.\mathbf{A} \;\hat{=}\; \inf_{v\in\mathsf{itr}.\mathbf{A}} \left(\limsup_{n\to\infty} \frac{1}{n} \sum_{m=1}^{n} v.m \right) \; .$$

The optimal value $\mathsf{val}.\mathbf{A}$ corresponds to the best possible behaviour of $\mathbf{A}$ as opposed to the best guaranteed. The difference between the infimum in the definition of val and the one in (5.4) is that it includes the initial densities of $\mathbf{A}$. This is expressed in the following theorem.

Theorem 5.4 Let $\mathbf{A} = (\Gamma, I, P)$ be live. Then

$$\mathsf{val}.\mathbf{A} \;=\; \min_{f\in I} f * \mathsf{opt}.(\mathsf{proc}.\mathbf{A}) \; .$$

So the main difference between $\mathsf{val}.\mathbf{A}$ and $\mathsf{opt}.(\mathsf{proc}.\mathbf{A})$ is that $\mathsf{val}.\mathbf{A}$ might choose an initial density that minimises $\mathsf{opt}.(\mathsf{proc}.\mathbf{A})$ should it be a non-constant function.

We note that val is a monotonic performance measure, as stated in the following corollary.

Corollary 5.5 $\mathbf{A} \sqsubseteq \mathbf{C} \;\Rightarrow\; \mathsf{val}.\mathbf{A} \leq \mathsf{val}.\mathbf{C}$.

Moreover, we know from theorem 5.4 that the average-cost optimal value of a system can be calculated by solving the corresponding problem for the associated Markov decision process. We have not yet used the information about the optimal policy that corresponds to the optimal value. As a matter of fact there is always such a stationary optimal policy [98]. This in turn corresponds to a deterministic probabilistic action system. Corollary 5.6 records this.

Corollary 5.6 Let $\mathbf{A}$ be a live probabilistic action system. Then there exists a deterministic system $\mathbf{C}$ that refines $\mathbf{A}$, and satisfies $\mathsf{val}.\mathbf{C} = \mathsf{val}.\mathbf{A}$.

A Markov decision process $\mathbf{M} = (\Gamma, P)$ is called *connected* if for all $\tau, \tau' \in \Gamma$ there is a number $n \in \mathbb{N}$ such that

$$\tau' \;\in\; \bigcup\{\mathsf{car}.f \mid f \in (\Downarrow P)^n.\tau\} \;.$$

The term $(\Downarrow P)^n$ describes the n-th iteration of the probabilistic state relation $\Downarrow P$ corresponding to P. Thus, $f \in (\Downarrow P)^n.\tau$ means that probabilistic state f is reachable after n steps starting in state τ. Taking the carrier $\mathsf{car}.f$ of f yields the states having positive probability in f. The union of the carriers $\bigcup\{\mathsf{car}.f \mid \ldots\}$ describes the set of all states τ' reachable after n steps starting from state τ. Summarised, a Markov decision process is connected if all states τ, τ' are reachable from each other in a finite number of steps.

If a Markov decision process $\mathsf{proc}.\mathbf{A}$ is connected then $\mathsf{opt}.(\mathsf{proc}.\mathbf{A})$ is a constant function (i.e. the initial state is irrelevant) [98], say $(\lambda\,\tau : \Gamma \bullet x)$, and consequently $\mathsf{val}.\mathbf{A}$ equals x. If $\mathsf{proc}.\mathbf{A}$ is connected we call $\mathbf{A}$ *connected* as well. Connected systems are important because there are efficient algorithms to compute $\mathsf{val}.\mathbf{A}$ if $\mathbf{A}$ is connected[1]. Connectedness of a system can be model checked efficiently using the Fox-Landi algorithm [34, 98]. It is not preserved by cost refinement as the following example shows:

Example 5.7 Figure 5.1 shows two equivalent probabilistic action systems. System A switches between the two states $x = \mathsf{false}$ and $x = \mathsf{true}$ incurring a cost of 2, if x equals true. System B incurs a cost of 1 during each transition never changing the state. Both systems are initially with equal probability in state $x = \mathsf{false}$ or state $x = \mathsf{true}$. System A is connected. System B is

[1]In [98] connected systems are referred to as communicating. We stick to the graph theoretical naming used in [26].

```
system A                          system B
variables                         variables
  x : 𝔹;                            x : 𝔹;
initialisation                    initialisation
  x := true ½⊕ x := false;          x := true ½⊕ x := false;
actions                           actions
  cycle =                           cycle = | 1.0 |;
    (| x |; | 2.0 | ⊔ | ¬ x |);   end
    x := ¬ x;
end
```

Figure 5.1: Two equivalent systems

not connected since, for example, $x = \mathsf{true}$ is not reachable from $x = \mathsf{false}$. Observe also that systems A and B have no policy in common.

The equivalence of the two systems can be established using theorem 4.7 with the following probabilistic simulation (see the appendix for the proof):

$$sim \quad = \quad x := \mathsf{true}\ {}_{\frac{1}{2}}\!\oplus\ x := \mathsf{false}\ .$$

In general, assuming **A** is a connected system, and using the software from chapter 6 we have computed an optimal implementation. If **A** was refined by an unconnected system **B**, it could well be that for the refined system **B** it is impossible to compute an optimal implementation. Indeed, the memory requirements for dealing with unconnected systems are very high (see chapter 6). There is also a positive side to this problem: If we were only interested in the optimal value of **B**, we could compute it using **A** as input of the software tool.

We close this section with a remark on average-cost optimality that serves to characterise its nature further. Proposition 5.8 can be found in similar form in [110], proposition 7.1.1.

Proposition 5.8 Let v be an infinite trace, and $N \geq 1$ fixed. Then:

$$\limsup_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} v.i \quad = \quad \limsup_{n\to\infty} \frac{1}{n-(N-1)} \sum_{i=N}^{n} v.i\ .$$

Proposition 5.8 states a property that one would expect from the average cost of an infinite trace of some system. The cost accumulated over an initial finite period of time does not influence the overall average cost of a trace.

5.4 Example: Polling System

The two probabilistic action systems *POLLING* and *REDPOLL* from chapter 4 are both live and connected. As mentioned above model checkers can

STATIONS	*CAPACITY*	val.**REDPOLL**
5	2	7.7746
6	2	9.7468
7	2	11.7312
8	2	13.7222
9	2	15.7170

Table 5.1: Values of some instances of system *REDPOLL*

be used to verify connectedness. The software tool DYNAS described in chapter 6 automatically checks for liveness of a system. Although we occasionally refer to the software tool in this section knowledge of the tool is not required. No reference is made to particular steps involved in using the tool. Note, however, that all calculations necessary to derive an optimal implementation have been solved automatically by the tool. In chapter 6 we discuss the software tool in detail.

The cost structure of system *POLLING* is set up so that small queues are preferred as well as stationarity of the server. Each packet waiting in some buffer causes a cost of one. A cost of one is also incurred each time period during which the which the server is walking. Serving itself causes no cost. Using state function *sim* (defined on page 59) this can easily be transferred to system *REDPOLL*. The only difference is that instead of the walking server we deal with walking stations.

We seek an optimal implementation for system *POLLING*. Because the systems are equivalent we only need to analyse system *REDPOLL* which has a much smaller state space. We calculate optimal implementations for systems with $CAPACITY = 2$ and a variety of values of *STATIONS* to increase our confidence in the resulting optimal system. The optimal average-cost values and optimal implementations have been calculated using the software tool DYNAS which is the subject of chapter 6. We think that it would be difficult to guess the optimal guard gd_{serve} of action *serve* below.

We derive an optimal implementation for system *REDPOLL* with parameters $STATIONS = 5$ and $CAPACITY = 2$. All we need to find is a guard $\mid gd_{serve} \mid$ for action *serve*. We can use its negation as guard in action *walk*. We define:

$$gd_{serve} \;\widehat{=}\; loc.1 = 0 \Rightarrow gd_0 \wedge gd_1 \wedge gd_2 \;.$$

The predicates gd_0, gd_1, and gd_2 are defined and explained below. From the precedent $loc.1 = 0$ of implication gd_{serve} we gather that the optimal server always serves when the buffer at station one is not empty. Otherwise one of the three conditions in its antecedent must hold. The predicates treat the three possible values of $loc.2$.

$$gd_0 \;\widehat{=}\; loc.2 = 0 \Rightarrow$$

$$
\begin{aligned}
&loc.3 > 0 \Rightarrow \\
&\quad (loc.3 = 1 \Rightarrow loc.4 = 0 \vee loc.5 = 0) \wedge \\
&\quad (loc.3 = 2 \Rightarrow loc.4 = 0)
\end{aligned}
$$

If $loc.2$ equals 0, then the server serves if the buffer at station three, $loc.3$, is also empty. Otherwise, if $loc.3 > 0$, we distinguish the two remaining cases for $loc.3$. Observe that the server is more inclined to serve when $buf.3$ is full. This is because rejecting packets does not cause any cost. Remember that the system incurs a cost of 1 for walking from station to station per unit of time.

$$gd_1 \;\widehat{=}\; loc.2 = 1 \Rightarrow loc.3 = 0 \wedge loc.4 = 0$$

If one packet waits in the buffer of station two, the server serves if no packets are waiting at the two immediately following stations.

$$gd_2 \;\widehat{=}\; loc.2 = 2 \Rightarrow loc.3 = 0 \wedge loc.4 = 0 \wedge loc.5 = 0$$

If the buffer of station two is full, all three buffers not mentioned in the guard must be empty. Otherwise the serve walks. This differs from the way station three is treated. The reason is that it is much cheaper to get to station two than to station three. We now present the actions *serve* and *walk* of the optimal system *OPTPOLL*. The rest of the specification is identical to system *REDPOLL*. Action *OPTPOLL.serve* simply adjoins gd_{serve} to the existing guard:

$$
\begin{aligned}
&OPTPOLL.serve \;=\; \\
&\quad |\, \textstyle\sum s : STATION \bullet \$(loc.s) \,|; \\
&\quad |\, \neg\, mov \wedge gd_{serve} \,|; arrival; departure;
\end{aligned}
$$

The guard of action *OPTPOLL.walk* is the negation of $|\, \neg\, mov \wedge gd_{serve} \,|$,

$$
\begin{aligned}
&OPTPOLL.walk \;=\; \\
&\quad |\, 1.0 \,|; \\
&\quad |\, \textstyle\sum s : STATION \bullet \$(loc.s) \,|; \\
&\quad |\, mov \vee \neg\, gd_{serve} \,|; arrival; \\
&\quad (mov := \mathsf{true} \;{}_{0.5}\!\oplus\; (mov := \mathsf{false} \parallel rotate));
\end{aligned}
$$

As shown in table 5.2 system *OPTPOLL* is also an optimal implementation of the pictured systems with six, seven, eight and nine stations. This means in systems with more than five stations packages waiting at stations not mentioned in the guard have no influence on the behaviour of the optimal server. The reason is that the server expects a packet to arrive at a near station. Compared to the cost of waiting travelling to a far away station would appear expensive.

STATIONS	*CAPACITY*	val.**OPTPOLL**
5	2	7.7746
6	2	9.7468
7	2	11.7312
8	2	13.7222
9	2	15.7170

Table 5.2: Corresponding values of system *OPTPOLL*

System *POLLING* contains explicit cost statements to specify the cost associated with packets waiting in some buffer, and the cost associated with moving between stations. It is not so obvious that, in fact, there is an implicit cost objective specifying that rejection of packets is free. Although the original model *POLLING* has all operational features that we would expect of a finite-buffer system, it might not express our performance objectives properly. If we wanted to associate a cost of, say, one with each rejection we would need unbounded counters to represent the number of rejections. This cannot be expressed in our formalism. The probabilistic action system formalism can potentially cope with infinite state spaces. In [110] algorithms are presented to solve optimisation problems of this kind for system with countable state spaces. The algorithms work by solving a sequence of finite systems. This is an extension of our work which we leave for further research.

It is useful to seek an informal justification for why an optimal solution generated by the tool is optimal. Doing this one has to uncover the effect of performance objectives that have led to the optimal solution. Afterwards it is possible to validate whether the original model is appropriate.

5.5 Remarks

We keep refinement and optimality separate. Refinement as we have defined it is compatible with any optimality criterion. If refinement were to require knowledge of optimal implementations, one would need separate notions of refinement for the different optimality criteria.

We have not defined a performance measure for non-live systems. The behaviour of these systems is entirely described in terms of infinite traces. This suggests we use only infinite traces as a behavioural semantics. However, we think impasses are useful in their own right. Firstly, systems that are not live have a meaning. Liveness is a property of a system. There is no need to prove that something is a system, as would be the case if we were only to admit live systems. We also prefer to have proper vocabulary to characterise liveness, i.e. a system having no impasses. Secondly, the

semantics based on traces and impasses is instrumental in the soundness proofs of the refinement rules given in theorems 4.4 and 4.7 in the preceding chapter.

We represent performance objectives by expected costs that arise during the operation of the specified system. An implementation of such a specification is optimal if it minimises the costs incurred during operation. In some cases the use of rewards instead of costs will appear more natural. Using rewards, optimal implementations correspond to maximised rewards. Indeed, some literature is based on rewards [98] and some on costs [110]. Generally the concepts are not mixed, and one decides either to use the cost approach or the reward approach. One way to represent costs and rewards in a single model is to treat costs as negative rewards [110]. We take the same approach as [110] and use only non-negative costs. This does not restrict the method but simplifies the presentation. A sketch of how to transform a model containing negative costs, i.e. rewards, into one that contains only non-negative costs follows. Let $P : \Gamma \leftrightarrow (\mathbb{R} \times \mathbb{D}\,\Gamma)$ be an extended program with negative expected costs, and let $b = \mathsf{min}.\{c \in \mathbb{R} \mid (\exists\, \tau, f \bullet P.\tau.(c, f))\}$. Program P induces a Markov decision process (Γ, Q) where Q is given by

$$Q.\tau.(c, f) \;\; = \;\; P.\tau.(c + b, f) \; .$$

The optimal average costs of the processes (Γ, P) and (Γ, Q) are related by:

$$\mathsf{avg}.(\Gamma, P) \;\; = \;\; \mathsf{avg}.(\Gamma, Q) - b \; .$$

In the probabilistic predicate transformer model presented in [81] three kinds of choice are introduced: demonic, angelic and probabilistic choice. This is done to arrive at a general theory like [6] for non-probabilistic predicate transformers. A similar approach could have been undertaken here. There is a suitable theory of competitive Markov decision processes [32]. The competitive model assumes that there are two competing players. One player tries to maximise the cost incurred; the other tries to minimise it. Optimally an equilibrium could be achieved where minimum and maximum are equal. In general, this equilibrium does not exist though. Another problem with the competitive model is that it is not appropriate to model the systems we are interested in. The only player involved is the controller trying to act optimally in a probabilistic environment. For some basic game theoretic notions consult [89].

We only regard deterministic policies. More general randomised policies have been investigated [28] but it can be shown general optimisation problems always have optimal solutions among deterministic policies [28]. This is not true anymore if additional constraints (i.e. state-action frequencies) are included [28].

The program model in [48] is operational like ours but does include an improper state “$\perp$” to treat divergence. It can be embedded into the probabilistic predicate transformer logic of [88]. Residual probability, i.e. the

probability of the occurrence of "$\perp$", in those models leads to divergence where in our model it leads to impasses. Our performance measure requires liveness (as defined in section 4.4), which makes it important to be able to express impasses. In fact, the role impasses play in our semantical model is similar to the role divergence plays in other models. In the extreme case using trace refinement without impasses would allow us to refine any dynamic system by a system that can only engage in the empty trace. That is it can not even start. Because we are especially interested in dynamic systems that are in continual operation impasses are considered ill-behaviour. Hence new impasses should never be introduced in a refinement. We have chosen not to introduce a special blocking state "$\top$", because we are not interested in the probability of reaching an impasse but only the possibility of reaching an impasse. If f is a probabilistic state on $\Gamma \cup \{\top\}$, then we call the restriction of f to Γ a sub-probability density, or short sub-density [71]. By allowing sub-densities instead of probabilistic states everywhere in our vector model, and modifying the definition of sequential composition by removing the condition $\mathsf{car}.f \subseteq \mathsf{dom}.G$, we would arrive at a model where the probability of the occurrence of $\top$ equals $1 - \sum_{\tau \in \Gamma} f.\tau$. Assume further that also our trace model were to use sub-densities. Then some infinite sequence of sub-densities f^∞ of a system $\mathbf{A}$ might have the property $\lim_{n\to\infty} \sum_{\tau\in\Gamma} f^\infty.n.\tau = 0$. As an immediate consequence $\lim_{n\to\infty} \sum_{\tau\in\Gamma} v.n.\tau = 0$ would also hold, where v is an infinite cost trace corresponding to f^∞. Hence, $\mathsf{val}.\mathbf{A} = 0$. This is not what we intended since now, in general, behaviour that leads to impasses is favoured over continual behaviour. Our model does not use sub-densities explicitly but identifies all of them by entirely blocking progress (through $\mathsf{car}.f \subseteq \mathsf{dom}.G$) if one were to occur in the alternative model.

Chapter 6

DYNAS

In this chapter we describe our effort to implement a software tool that computes an optimal implementation of a probabilistic action system. The software tool is implemented in the C programming language, Java [18, 114], JavaCC [63], and MySQL [90]. MySQL features a C-API, and a JDBC package is available as well. The software runs on Debian GNU/Linux [27]. For presentation purposes we use a functional programming notation similar to ML [95]. We refer to the software tool as DYNAS. We needed a name.

DYNAS consists of four separate component programs: the compiler, the expander, the solver, and the printer. If they are executed in that order, then an optimal implementation is computed for a probabilistic action system. Figure 6.1 pictures a schematic of the operation of DYNAS. The input is a syntactic probabilistic action system $\mathbf{A}$. This is translated into an abstract syntax tree $\mathbf{T}$ by the compiler. The expander computes the semantics of $\mathbf{T}$ and generates the corresponding Markov decision process $\mathsf{proc}.\mathbf{A}$. Next the solver takes the Markov decision process $\mathsf{proc}.\mathbf{A}$ and computes an optimal stationary policy Φ of $\mathsf{proc}.\mathbf{A}$. Finally, the optimal policy is translated into

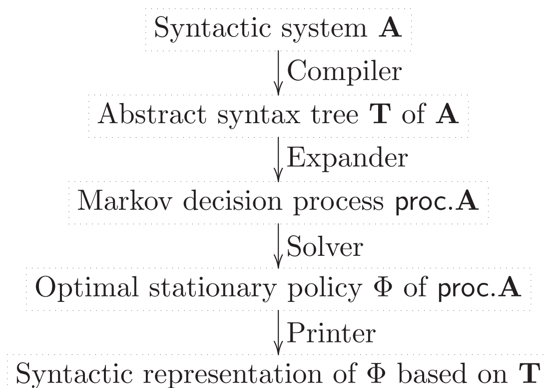


Figure 6.1: Operation of DYNAS

a textual representation by the printer. The textual representation relates to the input system **A** by using some information contained in the abstract syntax tree **T**.

The following sections 6.1 to 6.4 describe the function and implementation of the respective component programs.

6.1 Compiler

A probabilistic action system as accepted by the compiler is a plain text file. Similar to B, it uses the ASCII character set. Figure 6.2 shows system *REDPOLL* in ASCII format. See appendix C for a description of the ASCII-symbols used. Note that the ASCII-representation uses the function \$ to cast natural numbers into real numbers. In the formal reasoning in this chapter we do not use the ASCII-format of specifications though. Instead we use the symbolic representation to improve readability.

The compiler generates an abstract syntax tree from the input and writes it to the database. It is implemented in Java and JavaCC. Java objects which make up the nodes of the tree are mirrored into tables in the database. Upon writing each object is assigned a unique integer number as identifier, called *object key*. These object keys are used elsewhere to refer to locations in the specification. Custom sets C are represented as integer ranges $0 \mathinner{..} (\mathsf{card}.C - 1)$. So the expander may treat them like natural numbers.

Type-checking is based on the rules described in the B-Book [2]. A set of rewrite rules is used together with a simple form of unification. All type-checking is performed while the abstract syntax tree is constructed. The basic types known to the compiler are natural numbers $\mathbb{N}$, real numbers $\mathbb{R}$, boolean values $\mathbb{B}$, and custom sets C introduced in a specification. If the value of a variable x is restricted to a subset of one of these sets, then this is checked dynamically by the expander. When the expander encounters an assignment $x := e$ it checks whether e is a member of the set $VALUE_x$. The set $VALUE_x$ describes the possible values of variable x declared in the variables section of a system. Two type constructors, power set $\mathbb{P}\,\Theta$ and product set $\Theta_1 \times \Theta_2$, are also given. All other types are derived from these. Relations, for instance, are given by: $\Theta_1 \leftrightarrow \Theta_2 = \mathbb{P}(\Theta_1 \times \Theta_2)$. Because functions and sequences are subsets of relations they are not distinguished from relations in this type system. The additional properties that functions and sequences must satisfy are checked by the expander as outlined above. Examples of rewrite rules follow. We have shaded symbols read from the input to make them easier to differentiate from the type constructors.

If a constant belongs to one of the basic types, that basic type is inferred.

```
system REDPOLL
constants
  STATIONS = 4; // number of stations
  CAPACITY = 2; // capacity of buffers

sets
  STATION = 1..STATIONS;
  ARRIVAL = { TRUE |-> 0.1, FALSE |-> 0.9 };
  NEXT = %s:1..STATIONS-1.( TRUE | s + 1 ) \/ { STATIONS |-> 1 };

variables
  buffer : STATION --> 0..CAPACITY; // buffers at stations
  moving : BOOL; // true if server is moving to next station

initialisation
  // initially the server is stationary and all buffers are empty
  moving := FALSE ||
  buffer := STATION >< {0};

programs
  arrival = // arrival of packets at stations
    ?S : pow( buffer~[ 0..CAPACITY-1 ] )
    |*!s:buffer~[ 0..CAPACITY-1 ].( TRUE | ARRIVAL( s : S ) ).
      buffer := buffer <+ %s:S.( TRUE | buffer(s) + 1 );

  departure = // departure of processed packets from stations
    [ buffer( 1 ) = 0 ]
    []
    [ buffer( 1 ) > 0 ];
      (buffer := buffer <+ { 1 |-> buffer(1) - 1 } [[ 0.25 ]] skip)

actions
  serve = [ +!s:STATION.( TRUE | $( buffer(s) ) ) ];
    [ not( moving ) ]; arrival; departure;

  walk = [ $1 + +!s:STATION.( TRUE | $( buffer(s) ) ) ];
    arrival;
    (
      moving := FALSE
      [[0.5]]
      moving, buffer := TRUE, %s:STATION.( TRUE | buffer( NEXT(s) ) )
    );
end
```

Figure 6.2: System *REDPOLL* as accepted by the compiler

Remember that C stands for a type of custom values, not some subset.

$$\frac{n}{\mathbb{N}}\ [\, n \in \mathbb{N}\,] \qquad \frac{b}{\mathbb{B}}\ [\, b \in \mathbb{B}\,] \qquad \frac{c}{C}\ [\, c \in C\,]$$

Using the following set of rules we can type arithmetical and set expression. In conjunction with the above we derive the type of $3+2$ as $\mathbb{N}$, and the type of $(2+5)\,..\,(17+2)$ as $\mathbb{P}(\mathbb{N})$.

$$\frac{\mathbb{N} + \mathbb{N}}{\mathbb{N}} \qquad \frac{C}{\mathbb{P}(C)} \qquad \frac{\mathbb{N}\ ..\ \mathbb{N}}{\mathbb{P}(\mathbb{N})}$$

Set expressions are combined by set the operators power set and Cartesian product, or some derived operator. The rewrite rule concerning the power set operator is defined as:

$$\frac{\mathbb{P}\ \mathbb{P}(\Theta)}{\mathbb{P}(\mathbb{P}(\Theta))}$$

It says that if one forms the power set of a set with type $\mathbb{P}(\Theta)$, then the type of the power set is $\mathbb{P}(\mathbb{P}(\Theta))$. Using the Cartesian product rule below we can derive the type of $1\,..\,2 \times C$ as $\mathbb{P}(\mathbb{N} \times C)$. The other two rules deal with the sequence operator and the relation operator:

$$\frac{\mathbb{P}(\Theta_1) \times \mathbb{P}(\Theta_2)}{\mathbb{P}(\Theta_1 \times \Theta_2)} \qquad \frac{\mathbb{P}(\Theta_1) \leftrightarrow \mathbb{P}(\Theta_2)}{\mathbb{P}(\mathbb{P}(\Theta_1 \times \Theta_2))} \qquad \frac{\mathrm{seq}\ \mathbb{P}(\Theta)}{\mathbb{P}(\mathbb{P}(\mathbb{N} \times \Theta))}$$

The following rules deal with relational operators: relational inverse, identity relation on a set, and first element of a sequence. We can derive the type of the expression $\mathsf{first}.(\mathrm{id}\,.(1\,..\,2))$ by making use of two of the rules. It is $\mathbb{N}$.

$$\frac{\mathbb{P}(\Theta_1 \times \Theta_2)\ \sim}{\mathbb{P}(\Theta_2 \times \Theta_1)} \qquad \frac{\mathsf{id}\ \mathbb{P}(\Theta)}{\mathbb{P}(\Theta \times \Theta)} \qquad \frac{\mathsf{first}\ \mathbb{P}(\mathbb{N} \times \Theta)}{\Theta}$$

Empty set and empty sequence are polymorphic objects. We introduce a type variable ϑ to express polymorphism. Wherever a type variable ϑ occurs any type expression can be inserted. So ϑ acts like place-holder for type expressions. The type of the empty set is the power set of some set ϑ; and the type of the empty sequence is the power set of the Cartesian product of the natural numbers and some set ϑ:

$$\frac{\varnothing}{\mathbb{P}(\vartheta)} \qquad \frac{\langle\,\rangle}{\mathbb{P}(\mathbb{N} \times \vartheta)}$$

$$
\begin{array}{lll}
\textbf{fun}\ \mathsf{unify}\ x & \vartheta & = x \\
\quad |\ \mathsf{unify}\ \vartheta & y & = y \\
\quad |\ \mathsf{unify}\ \mathbb{P}(x) & \mathbb{P}(y) & = \\
\qquad\qquad \textbf{if}\ \mathsf{unify}\ x\ y = \mathsf{fail}\ \textbf{then}\ \mathsf{fail}\ \textbf{else}\ \mathbb{P}(\mathsf{unify}\ x\ y) \\
\quad |\ \mathsf{unify}\ (x_1 \times y_1)\ (x_2 \times y_2)\ = \\
\qquad\qquad \textbf{if}\ \mathsf{unify}\ x_1\ x_2 = \mathsf{fail}\ \textbf{or}\ \mathsf{unify}\ y_1\ y_2 = \mathsf{fail} \\
\qquad\qquad\quad \textbf{then}\ \mathsf{fail} \\
\qquad\qquad\quad \textbf{else}\ (\mathsf{unify}\ x_1\ x_2) \times (\mathsf{unify}\ y_1\ y_2) \\
\quad |\ \mathsf{unify}\ _ & _ & = \mathsf{fail}
\end{array}
$$

Figure 6.3: Unification of type expressions

The value of type variables must be inferred at compilation time. All expressions must be fully-typed. Inference of types is achieved by unification. Figure 6.3 shows the unification function we use. If two terms Θ_1 and Θ_2 cannot be unified, then function unify yields fail. We use fail as a special type constant to express failure of unification. Roughly speaking Θ_1 and Θ_2 can be unified if they are structurally similar, except where type variables occur. These type variables are replaced by matching type expressions. The returned type expression, if it exists, is the most general unifier of the two type expressions Θ_1 and Θ_2.

We give two examples of rewrite rules that involve unification. The first one deals with set union. Using it we can infer that the type of $\langle\,\rangle \cup (\varnothing \times 1\,..\,2)$ is $\mathbb{P}(\mathbb{N} \times \mathbb{N})$.

$$
\frac{\mathbb{P}(\Theta_1)\ \boxed{\cup}\ \mathbb{P}(\Theta_2)}{\mathbb{P}(\Theta)} \quad [\ \mathsf{unify}\ \Theta_1\ \Theta_2 = \Theta \neq \mathsf{fail}\ ]
$$

The second rule treats the append operation on sequences. It allows us to infer that the type of $\langle\,\rangle \leftarrow 1$ is $\mathbb{P}(\mathbb{N} \times \mathbb{N})$.

$$
\frac{\mathbb{P}(\mathbb{N} \times \Theta_1)\ \boxed{\leftarrow}\ \Theta_2}{\mathbb{P}(\mathbb{N} \times \Theta)} \quad [\ \mathsf{unify}\ \Theta_1\ \Theta_2 = \Theta \neq \mathsf{fail}\ ]
$$

A type expression Θ resulting from unification may still contain type variables. This means some type information is unspecified in Θ. To enforce that all type information is available we need to check whether Θ contains no occurrence of ϑ. Such a type expression Θ is called *ground*. Figure 6.4 shows a function ground that computes if a type expression Θ is ground. Note that the wild-card _ in the last clause of the function covers all the basic types $\mathbb{N}$, $\mathbb{B}$, etc. It also treats fail so that the two functions unify and ground can be nested easily.

$$\begin{aligned}
&\textsf{\textbf{fun} ground } \vartheta &&= \textsf{false}\\
&\quad|\ \textsf{ground fail} &&= \textsf{false}\\
&\quad|\ \textsf{ground } \mathbb{P}(x) &&= \textsf{ground } x\\
&\quad|\ \textsf{ground } (x \times y) &&= \textsf{ground } x \textsf{ \textbf{and} ground } y\\
&\quad|\ \textsf{ground } _ &&= \textsf{true}
\end{aligned}$$

Figure 6.4: Ground type expressions

Equality and set membership both require ground type expressions. By their application type information regarding the two expressions Θ_1 and Θ_2 is dropped. Type $\mathbb{B}$ is inferred:

$$\frac{\Theta_1 = \Theta_2}{\mathbb{B}} \quad [\ \textsf{ground}\,(\textsf{unify}\,\Theta_1\,\Theta_2)\ ]$$

Observe that a rewrite rule similar to set membership must be used to type variables x, which are specified in the form $x : \mathbb{P}(\Theta)$. When variable x is encountered in an expression, type Θ is inferred.

$$\frac{\Theta_1 \in \mathbb{P}(\Theta_2)}{\mathbb{B}} \quad [\ \textsf{ground}\,(\textsf{unify}\,\Theta_1\,\Theta_2)\ ]$$

6.2 Expander

The input of the expander is an abstract syntax tree generated by the compiler (see section 6.1). This corresponds to a type-checked specification of a probabilistic action system **A**. The expander computes the semantics of **A** with state space reach.**A** by executing the action of **A** repeatedly. If system **A** is not live the expander aborts with an error message. Hence, if the expander terminates successfully, the resulting semantical system corresponds to a Markov decision process. Information generated while executing the action of **A** is included into the semantics. It is used to determine which syntactical implementations belong to particular semantical implementations.

6.2.1 Data Structures

All data are stored in a relational database. This is advantageous during debugging because the database can be queried. This often helps in locating program errors. Searching sequential files is much more difficult and usually involves writing special debugging tools. Another advantage of using a

relational database is that it provides fast insertion and search functionality. Implementation of a fast data storage and retrieval facility is a complex task. Within the time limits of our work it would have been impossible to tackle. The semantics of a probabilistic action system **A** is represented by three tables, STATE, INIT, and TRANS, such that

$$\mathbf{A} = (\mathsf{STATE}, \mathsf{INIT}, \mathsf{TRANS}) .$$

We present the three tables in SQL notation. Table STATE has three fields, KEY, HASH, and DATA. Field DATA contains the value of a state τ in sequentialised form. Field KEY is a unique key, generated by the expander, that identifies a state. To check if a state τ' has already been encountered the expander has to look up table STATE, and compare the DATA field of each entry τ with the sequentialised form of τ'. Because it would be very costly to do this directly, each state is assigned a 32-bit hash key HASH on which the comparison is performed (see [91] for the CRC-based hash function used). However the hash key is not unique. So the DATA fields must still be compared when hash keys collide.

```
table STATE (
    KEY integer,
    HASH integer,
    DATA BLOB,
  primary key(KEY),
  index(HASH)
)
```

The acronym BLOB abbreviates 'binary large object'. The data-type BLOB is generally used to store unstructured data in a database.

Table INIT has a single field PROB that holds a sequentialised probabilistic state. I.e. table INIT simply represents the set of initial densities.

```
table INIT ( PROB BLOB )
```

Table TRANS represents the action P of a probabilistic action system. Each entry in the table has a unique identifier KEY. It specifies a transition

$$\mathsf{INI} \mapsto (\mathsf{COST}, \mathsf{SUCC}) \tag{6.1}$$

which stands for $(\mathsf{COST}, \mathsf{SUCC}) \in P.\mathsf{INI}$. Hence INI is an initial state of P, and SUCC the successor probabilistic state. The real value COST is the associated cost. Note that SUCC is stored as a sequentialised form of probabilistic state. It is an alternating sequence of integer numbers τ_i and double values p_i, encoding the probabilistic state $\{\tau_1 \mathbin{@} p_1, \tau_2 \mathbin{@} p_2, \ldots, \tau_n \mathbin{@} p_n\}$. States are represented by integer numbers, the unique keys STATE.KEY

supplied by table STATE. Based on the game semantics presented in section 6.2.2 field PLAY stores the (sequentialised) play that is associated with a transition (6.1).

```
table TRANS (
    KEY integer,
    INI integer,
    COST double,
    PLAY BLOB,
    SUCC BLOB,
  primary key(KEY),
  index(INI)
)
```

The tables STATE and TRANS are a representation of a Markov decision process $\mathbf{M} = (\Gamma, P)$. The numerical algorithms of section 6.3, which are employed to compute the optimal solution of $\mathbf{M}$, use only the integer values representing the states and transitions. The optimal solution (of a connected system) is then simply the optimal value represented by a real number, and a set of integers that corresponds to the optimal deterministic implementation. However, each transition (6.1) has also a label PLAY attached to it. Having marked a collection of transitions optimal, we also get a collection of plays that are optimal. From these plays a syntactical form of the optimal implementation can be derived (see section 6.4).

Ordering and Sequentialisation

To make access to mathematical objects more efficient all objects of identical type are ordered linearly. So internally ordered trees can be used to store sets making their retrieval efficient. Also, based on the order any two objects can be sequentialised and then compared for equality more efficiently. This is how sequentialised state is stored in the database.

In the following we sketch the ordering used. Remember, it is assumed that a system passed to the expander is type-checked. Natural numbers have the obvious order. For boolean numbers, false is smaller than true. Custom values inherit their order from the representing natural numbers. Pairs are ordered lexicographically. Sets A and B are ordered with respect to their cardinality and the order of their elements: $A \leq B \Leftrightarrow \mathsf{card}.A \leq \mathsf{card}.B \wedge (\mathsf{card}.A = \mathsf{card}.B \wedge A \neq B \Rightarrow \mathsf{min}.(A \setminus B) \leq \mathsf{min}.(B \setminus A))$. Set comparison can be efficiently implemented by enumerating the elements, and comparing the first distinct value found. The size of the sets is checked in advance.

Computing the Semantics of Syntactic Systems

We outline the algorithm used to achieve a certain level of completeness in the presentation of implementation issues. The algorithm consists of an initialisation phase followed by a loop:

First the set of initial probabilistic states I is computed from the program $SYS.\mathsf{initialisation}$. If I is empty expansion is aborted because system SYS is not live. Otherwise the initial probabilistic states are stored in the database.

Afterwards the actions act of system SYS are expanded. For each state τ reached compute the set of successor probabilistic states X_{act} of τ for all actions act. If, for some state, all sets X_{act} are empty system SYS is not live and expansion is aborted. Otherwise corresponding transition data is stored in the database.

Computations involving program semantics of initialisation and actions of syntactic systems are described in detail in section 6.2.2.

6.2.2 Game Semantics

It is impractical to compute the semantics of a program using the relational semantics of chapter 3. Instead we use an operational semantics in which a program is represented by a game tree. In the context of relational semantics $\mathcal{E}(\Gamma)$ we interpret a program P as an expression over $\mathcal{E}(\Gamma)$. This is different from the approach we take in game semantics. There a program P is a syntactical description of a game tree T. The relationship between P and T is established by the algorithm exec shown in figure 6.5. Finally, the game semantics and the relational semantics of P are linked by the algorithm squash in figure 6.6. They are equivalent. The meaning of non-probabilistic programs can be described by a similar kind of game semantics [9, 55]. They establish an equivalence between game semantics and predicate transformer semantics. Observing that a Markov decision process can be viewed as an infinite game [32], we follow the construction of [9, 55]. The game trees we use in the semantics are based on probabilistic games [89].

We do not consider all language constructs introduced in chapter 3. Finite nondeterministic choice and finite probabilistic choice are left out. They are similar to their binary counterparts thus do not provide additional insights. Programs are described by the following syntax:

$$\begin{aligned} P \;=\;& \Lambda \mid \\ & \mathsf{skip} \mid x := e \mid \mid q \mid \mid \mid r \mid \mid \\ & P_1 \parallel P_2 \mid P_1 ; P_2 \mid P_1 \sqcup P_2 \mid P_1 \;{}_p\!\oplus P_2 \;. \end{aligned}$$

The empty program Λ is introduced for technical reasons. The two programs $\mid q \mid$ and $\mid r \mid$ denote the guard and cost statements, respectively. It signals termination of basic program constructs, and must not occur in actual programs. We impose the following structural restriction on programs P: If

$P = P_1; P_2$ or $P = P_1 \parallel P_2$, and P_1 contains a probabilistic choice, then P_2 does not contain nondeterminism. This is necessary because the game semantics is only sound if algebraic laws (L17) and (L19) can be applied.

Game Trees

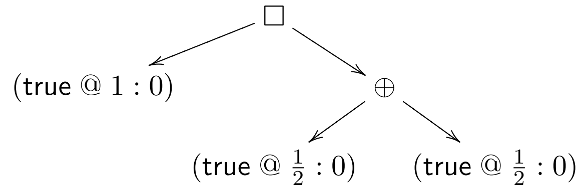
The game semantics of a program P is a binary tree. There are two types of inner nodes in a game tree corresponding to the two players: *choice* $\Box$ and *chance* $\oplus$. Leaves of a game tree are either proper states $(\tau \mathbin{@} p : c)$, or they indicate blocked execution (null). A game is played by following the structure of the game tree from the root. At each node encountered the appropriate players makes a move. If a choice node is encountered, player $\Box$ chooses a branch that is not null, and the game continues from there. If a chance node is encountered, player $\oplus$ chooses all branches with positive probability. The game continues in all branches (which must all be different from null). If a leaf $(\tau \mathbin{@} p : c)$ is encountered, this means that state τ occurs with probability p, and at a cost of c. Note that the same state may occur in more than one leaf. Playing a game corresponds to selecting a subtree of a game tree. This subtree is called a *play*. It is a game tree where all choice nodes have the form $\mathsf{tree}_2 \Box \mathsf{null}$ or $\mathsf{null} \Box \mathsf{tree}_1$.

$$\mathsf{tree} \;=\; \mathsf{null} \mid (\tau \mathbin{@} p : c) \mid \mathsf{tree}_1 \Box \mathsf{tree}_2 \mid \mathsf{tree}_1 \oplus \mathsf{tree}_2 \;.$$

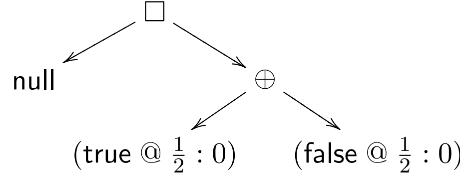
In the above description we have made two assumptions about nodes in a game tree. We say a game T is *well-formed* if: for each choice node $(\mathsf{tree}_1 \Box \mathsf{tree}_2)$ in T, either $\mathsf{tree}_1 \neq \mathsf{null}$ or $\mathsf{tree}_2 \neq \mathsf{null}$, and for each chance node $(\mathsf{tree}_1 \oplus \mathsf{tree}_2)$ in T, $\mathsf{tree}_1 \neq \mathsf{null}$ and $\mathsf{tree}_2 \neq \mathsf{null}$. The game semantics we use only produces well-formed games (see figure 6.5). Examples of programs and their game semantics follow.

The first example deals with the case where a program induces multiple games. These are programs where the game being played depends on the initial state.

Example 6.1 Let $P \in \mathcal{E}(\mathbb{B})$ be a program, $P = \lfloor b \rfloor \sqcup (b := \mathsf{true} \;{}_{\frac{1}{2}}\!\oplus \mathsf{skip})$. Program P gives rise to two games: one for initial state $b = \mathsf{true}$ and one for $b = \mathsf{false}$. We show true-game first:



The false-game of P is the following tree:

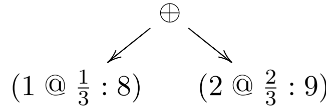


The second example demonstrates game semantics in the presence of cost statements. It relies on algebraic laws of section 3.6.

Example 6.2 Let $P = |\,7\,|; (x := 1 \;{}_{\frac{1}{3}}\!\oplus\; x := 2); |\,x\,|$. Program P is equivalent to a program where probabilistic choice does not occur within sequential composition:

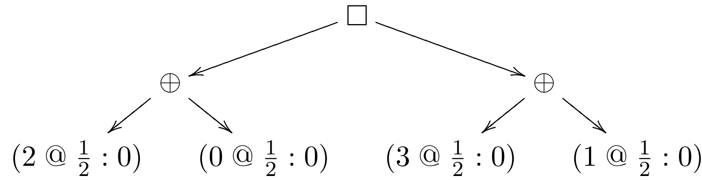
$$
\begin{array}{ll}
P & \\
= \; |\,7\,|; ((x := 1; |\,x\,|) \;{}_{\frac{1}{3}}\!\oplus\; (x := 2; |\,x\,|)) & (L17) \\
= \; (|\,7\,|; x := 1; |\,x\,|) \;{}_{\frac{1}{3}}\!\oplus\; (|\,7\,|; x := 2; |\,x\,|) & (L38)
\end{array}
$$

By way of this equivalence, program P induces the same game tree, below, for all initial states $n \in \mathbb{N}$:



The last example deals with the case of two consecutive choice statements.

Example 6.3 Let $P = (x := 1 \sqcup x := 2); (x := x + 1 \;{}_{\frac{1}{2}}\!\oplus\; x := x - 1)$. This program also does not depend on the initial state, thus, giving rise to only one game. The game tree associated with program P is:



To see this we transform P algebraically:

$$
\begin{array}{ll}
P & \\
= \; (x := 1; (x := x + 1 \;{}_{\frac{1}{2}}\!\oplus\; x := x - 1)) & (L11) \\
\quad \sqcup & \\
\quad (x := 2; (x := x + 1 \;{}_{\frac{1}{2}}\!\oplus\; x := x - 1)) & \\
= \; ((x := 1; x := x + 1) \;{}_{\frac{1}{2}}\!\oplus\; (x := 1; x := x - 1)) & (L16) \\
\quad \sqcup & \\
\quad ((x := 2; x := x + 1) \;{}_{\frac{1}{2}}\!\oplus\; (x := 2; x := x - 1)) &
\end{array}
$$

As in the preceding example 6.2 we have transformed P as long as there were choice statements within sequential compositions. In fact, the game semantics is defined this way.

Operational Semantics

Figure 6.5 pictures a functional definition of the game semantics of extended probabilistic state relations. The game tree $\mathsf{game}\,\tau\,P$ associated with program P and initial state τ is defined by:

$$\mathsf{game}\,\tau\;P \;\;=\;\; \mathsf{exec}\,\tau\;\tau\;1.0\;0.0\;P\;\langle\,\rangle\;.$$

The probability of initially being in state τ is 1.0. The associated initial cost is 0.0. And P is the program that remains to be executed. The role of the first two state parameters of exec and the last parameter, a stack of items of type rad, is less evident. The type of radicals, rad, is defined by:

$$\mathsf{rad} \;\;=\;\; (\tau; P) \mid (\tau \parallel P)\;.$$

We call $(\tau; P)$ a *sequential radical* and $(\tau \parallel P)$ a *parallel radical*. Radicals are used to perform the algebraic transformations of examples 6.2 and 6.3 while constructing the game tree of a program. The *radical stack* represents a branch in a tree that serves to compute the effect of a program consisting of assignments, sequential compositions and parallel compositions. These three program constructs modify the radical stack. When nondeterministic or probabilistic choices are encountered the radical stack is copied. We define the update $[\![\phi]\!]\,\rho$ of a radical stack ρ by a state function ϕ by

$$\begin{aligned}
[\![\phi]\!]\,\langle\,\rangle \;&\widehat{=}\; \langle\,\rangle\;, \\
[\![\phi]\!]\,((\tau; P) \to \rho) \;&\widehat{=}\; (([\phi]\,\tau); P) \to ([\![\phi]\!]\,\rho)\;, \\
[\![\phi]\!]\,((\tau \parallel P) \to \rho) \;&\widehat{=}\; (\tau \parallel P) \to ([\![\phi]\!]\,\rho)\;.
\end{aligned}$$

It updates intermediate states associated with sequential radicals. By doing this changes occurring on the left hand sides of sequential compositions become visible on the right hand sides.

Let τ be the current state in an execution of a program. For the moment assume P and Q are deterministic non-probabilistic programs. When a sequential composition $P; Q$ is encountered during the execution, radical $(\tau; Q)$ is pushed onto the stack and P is executed (line 10 in figure 6.5). Once P has been executed, yielding successor state $\hat{\tau}$, radical $(\hat{\tau}; Q)$ is popped from the stack (line 2). Then Q is executed using $\hat{\tau}$ as initial state. Assignments made in P are mirrored in sequential radicals on the stack (lines 8 and 9). So all modifications made by P are visible for Q. Similarly, when a parallel composition $(P \parallel Q)$ is encountered, execution continues with P (line 11). The radical $(\tau \parallel Q)$ is pushed onto the stack,

```
1  fun exec τ τ′ p c Λ        ⟨ ⟩               = (τ′ @ p : c)
2    | exec τ τ′ p c Λ        ((τ̂; P) → ρ)      = exec τ̂ τ′ p c P ρ
3    | exec τ τ′ p c Λ        ((τ̂ ∥ P) → ρ)     = exec τ̂ τ′ p c P ρ
4    | exec τ τ′ p c skip     ρ                 = exec τ τ′ p c Λ ρ
5    | exec τ τ′ p c ⌊ q ⌋    ρ                 =
6       if q.τ then (exec τ τ′ p c Λ ρ) else null
7    | exec τ τ′ p c ⌈ r ⌉    ρ                 = exec τ τ′ p (c + r.τ) Λ ρ
8    | exec τ τ′ p c (x := e) ρ                 =
9       exec τ ([x := e.τ] τ′) p c Λ ([[x := e.τ]] ρ)
10   | exec τ τ′ p c (P; Q)   ρ                 = exec τ τ′ p c P ((τ; Q) → ρ)
11   | exec τ τ′ p c (P ∥ Q)  ρ                 = exec τ τ′ p c P ((τ ∥ Q) → ρ)
12   | exec τ τ′ p c (P ⊔ Q)  ρ                 =
13      if exec τ τ′ p c P ρ = null and exec τ τ′ p c Q ρ = null
14        then null
15        else (exec τ τ′ p c P ρ) □ (exec τ τ′ p c Q ρ)
16   | exec τ τ′ p c (P a⊕ Q) ρ                 =
17      if exec τ τ′ (p ∗ a.τ) c P ρ = null or exec τ τ′ (p ∗ (1 − a.τ)) c Q ρ = null
18        then null
19        else (exec τ τ′ (p ∗ a.τ) c P ρ) ⊕ (exec τ τ′ (p ∗ (1 − a.τ)) c Q ρ)
```

Figure 6.5: Game semantics of programs $P \in \mathcal{E}(\Gamma)$

τ being the current state. After execution P yields a successor state τ'. Having executed P, radical $(\tau \parallel Q)$ is popped from the stack (line 3). Program Q is then executed with initial state τ too. The successor state of Q is merged with state τ' to yield the successor state of $(P \parallel Q)$. This is possible because programs P and Q cannot assign to a shared variable. By definition of parallel composition there cannot be a shared variable. Note that commutativity of parallel composition is preserved even though P is executed first.

If an execution encounters a program $(P_1 \sqcup P_2); Q$ then, as above, $(\tau; Q)$ is pushed onto the radical stack (line 10). Then P_1 and P_2 are executed separately with $(\tau; Q)$ at the top of their radical stack (lines 12 to 15). This means the program executed is $(P_1; Q) \sqcup (P_2; Q)$. Execution of probabilistic choice followed by sequential composition, $(P_1 \;{}_a\!\oplus\; P_2); Q$, is executed similarly (lines 16 to 19). This behaviour of function exec is based on the two algebraic laws (L11) and (L17). The same applies for parallel composition where laws (L13) and (L19) are used. Observe that all assignments encountered are also performed on state τ' (line 9). This way parameter τ' accumulates the final state of each execution branch encountered.

Given an initial state τ, a game tree T of a program P corresponds to a set of pairs of costs and probabilistic states X, such that $P.\tau = X$. The

$$
\begin{array}{ll}
\textsf{\textbf{fun} squash } (\tau \mathbin{@} p : c) & = \{(p * c, \{\tau \mathbin{@} p\})\} \\
\quad \mid \textsf{squash null} & = \varnothing \\
\quad \mid \textsf{squash } (x \mathbin{\Box} y) & = \textsf{squash } x \cup \textsf{squash } y \\
\quad \mid \textsf{squash } (x \oplus y) & = \textsf{squash } x + \textsf{squash } y
\end{array}
$$

Figure 6.6: Squashing a game tree

remainder of this section establishes this connection between game semantics and relational semantics.

Figure 6.6 pictures the function squash that transforms a game tree into a set of pairs of costs and probabilistic states. The addition of elements of $\mathbb{P}(\mathbb{R}_{\geq 0} \times (\Gamma \to \mathbb{R}_{\geq 0}))$, used in the last clause, is defined by

$$X + Y \;=\; \{(c + d, f + g) \mid (c, f) \in X \wedge (d, g) \in Y\} \;.$$

We also define scalar multiplication with a number $p \in (0, 1)$ by:

$$p * X \;=\; \{(p * c, p * f) \mid (c, f) \in X\} \;.$$

Intermediate probabilistic states that occur in function squash may add up to a total probability of less than one. We still use the notation for probabilistic states introduced earlier though.

Theorem 6.5 shows that squash $(\textsf{game}\,\tau\,P)$ is suitable to compute $P.\tau$. It is a direct consequence of lemma 6.4 which provides a more general result using function exec. It says that all subtrees of a game tree correspond to programs. We only have to take account of the initial probability of the subtree, and the initial cost which the subtree inherits from the containing game tree.

Lemma 6.4 Let $p \in (0, 1)$, $c \in \mathbb{R}_{\geq 0}$, τ an initial state, and P a program. Then:

$$p * (\mid c \mid ; P).\tau \;=\; \textsf{squash}\,(\textsf{exec}\,\tau\,\tau\,p\,c\,P\,\langle\,\rangle) \;.$$

The correctness of the composed function (squash o game) with respect to the relational semantics of programs follows:

Theorem 6.5 For a program P and initial state τ:

$$P.\tau \;=\; \textsf{squash}\,(\textsf{game}\,\tau\,P) \;.$$

Finally, observe that each play of $\textsf{game}\,\tau\,P$ corresponds to exactly one pair $(c, f) \in \textsf{squash}\,(\textsf{game}\,\tau\,P)$. This is easy to see because the union in squash is practically removed when the function is applied to a play. This yields the transition entries stored in table TRANS consisting of an initial state τ, a cost c, a probabilistic state f, and a corresponding (sequentialised) play (see section 6.4 for implementation details).

6.3 Solver

The solver reads the integer representation of the Markov decision process associated with the probabilistic action system from the database, and computes the average-cost optimal solution. On termination, the optimal solution, a vector of integer numbers representing transitions, is stored in the database. We have implemented three numerical algorithms that solve the average-cost optimisation problem (5.4): unichain policy iteration, multichain policy iteration, and value iteration. For more details on these and other related numerical algorithms see [98, 110]. In sections 6.3.1 and 6.3.2 we describe our experiences with the algorithms we have implemented. We use the abstract notion of Markov decision processes of chapter 5. It should be easy to relate to the data structures of the implementation presented in section 6.2.1.

6.3.1 Policy Iteration

Because it is known that an optimal policy is among the stationary ones [28] one needs only to consider stationary policies in the optimisation algorithm. Policy iteration searches through the space of stationary policies directly. This in contrast to value iteration where the search is based on the optimal value of a Markov decision process.

In the following let $\mathbf{M} = (\Gamma, P)$ be a Markov decision process. A stationary policy $(C, M) \in \mathsf{ipo}.\mathbf{M}$ is called *multichain* if the underlying state space Γ can be partitioned into sets Δ_0, Δ_1, Δ_2, ..., Δ_k, such that: set Δ_0 is *transient*, i.e. the probability of being in Δ_0 in the long run is zero, and the sets Δ_1, Δ_2, ..., Δ_k are not reachable from each other. Stationary policy (C, M) is called *unichain* if $k = 1$. The sets Δ_1, Δ_2, ..., Δ_k are called the *recurrent classes* of (C, M). A Markov decision process is called *multichain* if one of its stationary policies is multichain, and unichain otherwise.

The average cost of a stationary policy $(C, M) \in \mathsf{ipo}.\mathbf{M}$ can be characterised by the two equations [98]:

$$0 = (M - \mathsf{id}) * a \ , \tag{6.2}$$

$$0 = C - a + (M - \mathsf{id}) * z \ , \tag{6.3}$$

where $\mathsf{id} \in \mathcal{M}(\Gamma)$ is the identity program, i.e. $\mathsf{id}.\tau.\tau = 1$. Vector a is usually called the *gain* of policy (C, M) and equals the average cost $\mathsf{avg}.(C, M)$ of the policy. Vector z is usually referred to as *bias*. It is not determined uniquely by equation (6.3) but suitable side conditions exist to achieve uniqueness [98]. The following property holds:

$$z.\tau - z.\tau' = \limsup_{n\to\infty} \frac{1}{n} \sum_{m=1}^{n} (\mathsf{tot}.(C, M).\tau - \mathsf{tot}.(C, M).\tau')$$

It means that $z.\tau - z.\tau'$ is the average relative difference in total expected cost that results from starting in state τ instead of state τ'. Equation (6.2) says that states in the same recurrent class have the same gain. The average cost $\mathsf{avg}.(C, M)$ distributes the costs $C.\tau$ incurred in states τ of a recurrent class Δ_j evenly among all members of that class.

We distinguish multichain and unichain policy iteration algorithms according to the type of Markov decision process they can solve. Multichain policy iteration is treated, for instance, in [98]. In the following we describe the unichain policy iteration algorithm (see figure 6.7). For unichain Markov decision processes all associated average costs are constant functions. This implies that equation (6.2) is trivially satisfied. Unichain policy iteration proceeds as follows: First an arbitrary stationary policy (C, M) of a Markov decision process (Γ, P) is selected. For this policy, equation (6.3) is evaluated which yields the corresponding gain $a \in \mathbb{R}_{\geq 0}$ and bias $z \in \Gamma \to \mathbb{R}$. Note that a real value $a \in \mathbb{R}_{\geq 0}$ is used to represent the gain, and that $\dot{a}$ is the point-wise extension of a. In the next step an improved policy (D, N) is sought. Policy (D, N) is considered an improvement of policy (C, M) if it realises the minimum $\mathsf{min}_{(d,n)\in\mathsf{fun}.P}(d + n * z)$, but (C, M) does not. The expression $\mathsf{arg\,min}_{(d,n)} \ldots$ yields the argument (d, n) realising the minimum. If no improvement is possible the algorithm terminates. In this case a is the average-cost optimal value of (Γ, P) and (C, M) an average-cost optimal stationary policy. If on the other hand (D, N) is an improvement the algorithm continues with evaluation where (C, M) is replaced by (D, N).

The algorithm terminates with an optimal policy (C, M). In each improvement step a successor policy (D, N) is chosen such that either (C, M) is improved, or (D, N) equals (C, M). If (D, N) minimises the equation

$$D + N * z \ ,$$

then $D + N * z \leq C + M * z$ holds, again implying

$$D + N * z \leq \dot{a} + z \ . \tag{6.4}$$

If $(D, N) \neq (C, M)$ then (6.4) becomes a proper inequality. In this case one can show that either $\mathsf{avg}.(D, N) < \dot{a}$ or the bias of a transient state τ of policy (D, N) is smaller than $z.\tau$ [98, 115]. Only finitely many improvements are possible because there are only finitely many stationary policies. Upon termination the following equation is satisfied:

$$0 \ = \ \mathsf{min}_{(D,N)\in\mathsf{fun}.P}(D - \dot{a} + (N - \mathsf{id}) * z) \ . \tag{6.5}$$

It is called the average-cost optimality equation, and implies that $\dot{a}$ is the optimal value [98]. This in turn implies that (C, M) is the optimal stationary policy of (Γ, P).

Multichain policy iteration is capable of solving a very general class of Markov decision processes, where all stationary policies may be multichain.

input: Markov decision process (Γ, P)

var $M, N : \mathcal{M}(\Gamma)$;
var $C, D : \Gamma \to \mathbb{R}_{\geq 0}$;
var $a : \mathbb{R}_{\geq 0}$;
var $z : \Gamma \to \mathbb{R}$;

(initialise)
select $(D, N) \in \mathsf{fun}.P$ **arbitrarily**;

(iterate)
repeat
  $(C, M) := (D, N)$;
(evaluate)
  obtain a **and** z **by solving**
    $0 = C - \dot{a} + (M - \mathsf{id}) * z$;
(improve)
  choose
    $(D, N) \in \arg\min_{(d,n) \in \mathsf{fun}.P} (d + n * z)$
    setting $(D, N) = (C, N)$ **if possible**;
until $(C, M) = (D, N)$;

output: optimal value a and policy (C, M)

Figure 6.7: Policy iteration algorithm

Unfortunately the price in terms of memory requirements and time consumption is very high. We have managed to use it with systems that have up to about 2000 states. The memory available was 256 MB. By way of implementing multichain policy iteration we have also implemented the Fox-Landi algorithm [34, 98] to determine the recurrent classes of a Markov decision process. This algorithm can also be used to check if a Markov decision process is connected, i.e. all states are reachable from each other (see section 5.3). It is also possible to use linear programming instead of multichain policy iteration. This causes the same problems however [115]: it does not cope well with large state spaces. We have not implemented the linear programming algorithm to solve the Markov decision process.

Unichain policy iteration is only applicable if all stationary policies of the Markov decision process under consideration are unichain. This is a severe restriction, and this method has failed already in simple cases. The unichain property of a system also appears to be hard to prove considering that it is an artificial constraint imposed by the algorithm. Value iteration requires a much weaker property of the Markov decision process, that is also easier to check.

6.3.2 Value Iteration

A stationary policy (C, M) of a Markov decision process $\mathbf{M} = (\Gamma, P)$ is called *aperiodic* if there is a state function $L : \mathcal{M}(\Gamma)$ such that

$$\lim_{n\to\infty} M^n \;=\; L \;.$$

For instance, the following probabilistic state function M induces a stationary policy, say (C, M), that is not aperiodic:

$$M \;\widehat{=}\; \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \;.$$

Stationary policy (C, M) "oscillates". For all $n \in \mathbb{N}$:

$$M^{2n-1} \;=\; \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad M^{2n} \;=\; \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \;.$$

A Markov decision process $\mathbf{M}$ is called aperiodic if all stationary policies of $\mathbf{M}$ are aperiodic.

Remember that $\mathsf{fin}.\mathbf{M}.n$ is the optimal total cost of a process $\mathbf{M}$ after n transitions. Because a Markov decision process $\mathbf{M}$ has only finitely many policies of length n the infimum in (5.2) is attained. Hence $\mathsf{fin}.\mathbf{M}.n$ equals

$$\min_{(\kappa,\Phi)\in\mathsf{po}.\mathbf{M}.n} \mathsf{tot}.(\kappa,\Phi).n \;.$$

We remark that there may be no stationary optimal policy (κ, Φ) that minimises $\mathsf{tot}.(\kappa, \Phi).n$ [28, 110]. However, as n approaches infinity the significance of the past of system $\mathbf{M}$ as it evolves diminishes. We define a sequence of $z_n \;\widehat{=}\; \mathsf{fin}.\mathbf{M}.n$ of expected total costs accumulated till time instants $n \geq 0$, letting $z_0 \;\widehat{=}\; 0$. At time 0 no costs have been incurred. Sequence z_n equals (see e.g. [110]):

$$\begin{aligned} z_0.\tau &= 0 \;, \\ z_n.\tau &= \mathsf{min}_{(c,f)\in P.\tau}\; c + f * z_{n-1} \;. \end{aligned}$$

If $\mathbf{M}$ is aperiodic then $z_{n+1} - z_n$ approaches the optimal average cost $\mathsf{opt}.\mathbf{M}$ of process $\mathbf{M}$ [98, 110]. Note that the sequence z_n itself is unbounded and, as such, not suited for numerical computations. A simple and efficient method to achieve boundedness of sequence z_n is to replace it by a sequence of values that approximates $z_{n+1} - \mathsf{opt}.\mathbf{M}$ [98, 110]. To simplify the presentation of the value iteration algorithm that follows we use sequence z_n to compute successive approximations of $\mathsf{opt}.\mathbf{M}$.

Value iteration converges for aperiodic multichain Markov decision processes. In practice the aperiodicity requirement is not a problem because any stationary policy can be made aperiodic by a simple transformation

input: Markov decision process (Γ, P)

var $v, w : (\Gamma \to \mathbb{R})$;
var $a : \mathbb{R}_{\geq 0}$;
var $M : \mathcal{M}(\Gamma)$;

(initialise)
for $\tau \in \Gamma$ **do** $w.\tau := 0$ **end**;

(iterate)
repeat
  $v := w$;
  for $\tau \in \Gamma$ **do**
    $w.\tau := \min_{(c,f) \in P.\tau} (c + f * v)$
  end
until $(\mathsf{sp}\,(w - v) < \epsilon)$;

(finalise)
$a := \frac{1}{2} * (\max_\tau(w.\tau - v.\tau) + \max_\tau(w.\tau - v.\tau))$;
for $\tau \in \Gamma$ **do**
  $M.\tau := \arg\min_{(c,f) \in P.\tau} (c + f * v)$
end

output: optimal value a and policy (C, M)

Figure 6.8: Value iteration algorithm

[98, 110]. However a termination condition is only known for connected Markov decision processes [98]. Hence we require connectedness if a probabilistic action system is to be analysed with the software tool. As a matter of fact, it is sufficient to require weak connectedness, i.e. allow the system to have a nonempty transient class of states [98]. Connected processes $\mathbf{M}$ have constant optimal values $\mathsf{opt}.\mathbf{M}$, thus,

$$\mathsf{sp}\,(\mathsf{opt}.\mathbf{M}) \;=\; 0\;.$$

Function sp is the so-called span semi-norm on $\Gamma \to \mathbb{R}$ [98], defined by,

$$\mathsf{sp}\;v \;=\; (\max_{\tau \in \Gamma} v.\tau) - (\min_{\tau \in \Gamma} v.\tau)\;.$$

As sequence $z_{n+1} - z_n$ approaches $\mathsf{opt}.\mathbf{M}$ the span semi-norm of $z_{n+1} - z_n$ approaches 0. If $\mathsf{opt}.\mathbf{M}$ was non-constant $\mathsf{sp}\,(\mathsf{opt}.\mathbf{M})$ would equal some constant c. Because c is not known prior to calculation of $\mathsf{opt}.\mathbf{M}$, it can not be decided from the span semi-norm of the difference $z_{n+1} - z_n$ how close it is to $\mathsf{opt}.\mathbf{M}$.

Figure 6.8 pictures the value iteration algorithm. It consists of three stages. Initially the two vectors of real values v and w are set to zero. In the iteration to follow variables v and w correspond to the sequence values z_n and z_{n+1} respectively. Hence $w - v$ approximates the optimal value a of

M. The iteration stops when $\mathsf{sp}\,(w - v) < \epsilon$. The positive real value ϵ is a parameter of the algorithm specifying the accuracy of the solution. The solution is usually called ϵ-optimal [98]. One can show [98] that the sequence of differences $\mathsf{sp}\,(z_{n+1} - z_n)$ is monotonously decreasing

$$\mathsf{sp}\,(z_{n+2} - z_{n+1}) \;\leq\; \mathsf{sp}\,(z_{n+1} - z_n) \;,$$

and approaches zero as n approaches infinity

$$\lim_{n\to\infty} \mathsf{sp}\,(z_{n+1} - z_n) \;=\; 0 \;.$$

This implies that the algorithm terminates after finitely many iterations. Upon termination the following approximation holds

$$0 \;\approx\; C - \dot{a} + (M - \mathsf{id}) * z_n \;,$$

where a, C and M are determined in the finalisation part of the algorithm. Hence, if ϵ is small enough (C, M) is an average-cost optimal stationary policy.

We have implemented relative value iteration with a preceding aperiodicity transformation to achieve numerical stability and certain convergence of the algorithm [98]. It uses the bounded variant of sequence z_n mentioned above, and performs an aperiodicity transformation on the input Markov decision process.

Value iteration requires much less memory than policy iteration. Its memory requirements are linear in the size of the state space of the Markov decision process. For policy iteration these requirements are quadratic in the size of the state space. Value iteration is also much faster at each iteration [98]. From our experience value iteration takes many more iterations until termination. Policy iteration needs usually less than ten iterations, and value iteration a few hundred. Yet value iteration is still faster than policy iteration because each iteration takes much less time. Using value iteration we have solved systems with up to $100,000$ states and $500,000$ transitions (see section 7.6). Systems of this size are out of reach for the other numerical methods considered.

6.4 Printer

On termination the solver stores the optimal value and the optimal stationary policy computed in the database. The optimal policy is represented by the object keys TRANS.KEY that identify entries in table TRANS.

The printer reads the optimal value and the collection of object keys which determine the optimal policy. Then it reads the portion of the transition table TRANS to which the object keys refer.

action		guard
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 0\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 0\} \,\rvert$
walk	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 0, 3 \mapsto 0\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 0, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 0\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 2\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 2\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 2\} \,\rvert$
walk	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 2, 3 \mapsto 2\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 0\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 0, 3 \mapsto 2\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 0, 3 \mapsto 2\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 2\} \,\rvert$
walk	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 2\} \,\rvert$
walk	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 2, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 0\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 0\} \,\rvert$
walk	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 0, 2 \mapsto 2, 3 \mapsto 0\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 0, 3 \mapsto 1\} \,\rvert$
serve	if	$\lvert\, moving = \mathsf{false} \wedge buffer = \{1 \mapsto 2, 2 \mapsto 0, 3 \mapsto 0\} \,\rvert$

Figure 6.9: Optimal action of system *REDPOLL*

As can be seen in figure 6.5 choice and chance nodes in a game tree correspond to locations in the syntactic representation of a probabilistic action system. Note that because of the algebraic transformation taking place several nodes in a game tree T may correspond to the same syntactical location. The compiler has assigned object keys to the syntactical locations of $\sqcup$ and ${}_p\oplus$. These object keys are stored in the nodes of the game tree. The plays of game tree T inherit the object keys of T. Field TRANS.PLAY of table TRANS contains a sequentialised form of the plays including the object keys. Finite nondeterministic choice $\bigoplus$ and finite probabilistic choice $\bigsqcup$ have not been treated in section 6.2. Every branch of a finite choice corresponds to an index value. For finite choices $\bigoplus$ and $\bigsqcup$ index values are stored in the game tree in addition to the object key of $\bigoplus$, or $\bigsqcup$. All of this data is also

contained in the plays (and sequentialised plays).

The printer creates a text file, and writes the optimal value into it followed by a syntactical representation of the plays read from table TRANS. The output could be improved by combining the abstract syntax tree with the plays. This would yield a syntactical probabilistic action system. We leave this work open. We considered this programming task too complex to complete within the scheduled time. The implementation period for the entire software tool was limited to ten months.

Figure 6.9 shows the optimal action of system *REDPOLL* of figure 6.2, i.e. the polling system with 3 stations and a buffer capacity of 2. The optimal value is 3.9168. We have modified the output of the printer to improve readability. The game trees that occur in system *REDPOLL* do not contain chance nodes. The printer recognises this and does not print such game trees. They contain no useful information. The left column contains the name of an action, and the right column an enabling condition. This means the guard of action *serve*, say, of the optimal implementation is the disjunction of the corresponding rows of the table.

Chapter 7

Case Study: Lift System

The lift system investigated in this case study is similar to the one used in [2]. It consists of a number of parallel lifts and a number of floors which are served by the lifts. A similar system has also been analysed in [31] using the stochastic process algebra PEPA. We choose to examine the lift system because it has a high degree of nondeterminism and large state space for even moderate numbers of lifts and floors. The first half of the case study establishes a relationship between machines (see section 2.4) and probabilistic action systems. The relationship itself is kept informal. However it is based on an approach commonly used in systems engineering [41]. The second half of the case study is dedicated to analysing the initial specification of the lift system presented in section 7.2.5.

In detail, section 7.1 presents a machine describing the lift and its behaviour. In section 7.2 we show how the machine from section 7.1 can be used to derive a probabilistic action system describing the lift system. The resulting system is summarised in section 7.2.5. Unfortunately this probabilistic action system cannot be analysed using our software tool. Section 7.3 presents a transformation which results in a probabilistic action system that can be used with the software tool. It is applied in the following section 7.4. A by-product of the transformation is a substantial increase in the number of states of the system. In section 7.5 we use a fusion to reduce the size of the state space so that it becomes manageable by the software tool. Finally, in section 7.6 we present some results of our analysis of the lift system and discuss the material presented in this chapter.

7.1 State and Operation of the Lift Machine

We give a specification of a lift system in our action system formalism. It declares two constants *FLOORS* and *LIFTS*, the number of floors and the

number of lifts, respectively.

> machine SYS_LIFT
> constants $FLOORS$; $LIFTS$;
> constraints $FLOORS > 1 \land LIFTS > 1$;

The sets DIR, $LIFT$, and $FLOOR$ are used in the declaration of the variables. Lifts can move up or down. Correspondingly, set DIR defines the constants UP and DN, modelling the two possible directions. The lifts and floors of the system are represented by the sets $LIFT$ and $FLOOR$. The set $MOVE$ serves to define possible movements of the lifts. It computes the next position of a lift ll moving in direction dd. Its use simplifies the operational section of the specification. The domain of $MOVE$ corresponds to all admissible movements.

> sets
> $DIR = UP \mid DN$;
> $LIFT = 1 \mathinner{..} LIFTS$;
> $FLOOR = 1 \mathinner{..} FLOORS$;
> $MOVE =$
> $(\lambda\, ff : FLOOR, dd : \{DN\} \bullet (ff \neq 1 \mid ff - 1)) \cup$
> $(\lambda\, ff : FLOOR, dd : \{UP\} \bullet (ff \neq FLOORS \mid ff + 1))$;

The state of the lift system is defined by the variables *req*, *out*, *dir*, *floor*, and *moving*. Variable *req* models buttons outside the lift that are used to call a lift. Variable *out* model the buttons inside the lift used to select a destination floor. Variable *dir* contains the direction of moving lifts. Its value is meaningless for lifts that are not moving. Variable *floor* stores the location of all lifts. All lifts are considered to be at some floor. That includes moving lifts. Variable *moving* contains the set of lifts that are moving.

> variables
> $req : \mathbb{P}(FLOOR)$;
> $out : LIFT \leftrightarrow FLOOR$;
> $dir : LIFT \rightarrow DIR$;
> $floor : LIFT \rightarrow FLOOR$;
> $moving : \mathbb{P}(LIFT)$;

The initialisation section consists of a collection of parallel assignments. Initially, there are no open requests: $req = \varnothing \land out = \varnothing$. All lifts are stationary: $moving = \varnothing$. They are in the first floor: $floor = LIFT \times \{1\}$. The direction of all lifts is arbitrarily set to UP: $dir = LIFT \times \{UP\}$.

> initialisation
> $req, out, moving := \varnothing, \varnothing, \varnothing \parallel$
> $floor := LIFT \times \{1\} \parallel$
> $dir := LIFT \times \{UP\}$;

The actions section consists of five actions. Actions *reqLift* and *reqFloor* add requests to *req* and *out* respectively. Action *move* models the movement of lifts. Action *stop* stops a lift, and action *depart* occurs when a lift leaves a floor.

actions

If no stationary lift is located at floor *ff*, action $reqLift(ff)$ can occur. Floor *ff* is then added to *req*, requesting any lift to stop at floor *ff*. The set $LIFT \setminus moving$ denotes the stationary lifts, and the set $floor[LIFT \setminus moving]$ denotes the floors where at least one stationary lift is located.

$$
\begin{array}{l}
reqLift(ff : FLOOR) = \\
\quad |\ ff \notin FLOOR \setminus floor[LIFT \setminus moving]\ |;\ req := req \cup \{ff\};
\end{array}
$$

Action $reqFloor(ll, ff)$ can only occur if lift *ll* is not at floor *ff*. This adds the request for lift *ll* to stop at floor *ff*, $(ll \mapsto ff)$, to *out*.

$$
\begin{array}{l}
reqFloor(ll : LIFT, ff : FLOOR) = \\
\quad |\ ff \neq floor.ll\ |;\ out := out \cup \{ll \mapsto ff\};
\end{array}
$$

If $ll \in moving$ and it is physically possible to move lift *ll* in direction $dir.ll$, then action $move(ll)$ can occur. The lift *ll* is then moved one level in the given direction $dir.ll$. Note that a lift *ll* can move even if there are outstanding requests on floor $floor.ll$. Ultimately, we want to find out if and how movements and requests should be connected in an optimal implementation of system *SYS_LIFT*.

$$
\begin{array}{l}
move(ll : LIFT) = \\
\quad |\ ll \in moving \wedge (floor.ll, dir.ll) \in \mathsf{dom}.MOVE\ |; \\
\quad floor := floor \mathbin{\lhd\!\!\!+} \{ll \mapsto MOVE.(floor.ll, dir.ll)\};
\end{array}
$$

If $ll \in moving$, then lift *ll* can be stopped. Action $stop(ll)$ stops lift *ll* and removes answered requests from buffers *req* and *out*.

$$
\begin{array}{l}
stop(ll : LIFT) = \\
\quad |\ ll \in moving\ |; \\
\qquad moving := moving \setminus \{ll\}\ \| \\
\qquad out := out \setminus \{ll \mapsto floor.ll\}\ \| \\
\qquad req := req \setminus \{floor.ll\};
\end{array}
$$

Finally, action $depart(ll, dd)$ sets lift *ll* in motion facing direction *dd*. It requires that lift *ll* is stationary. It must also be possible for lift *ll* to move in direction *dd*.

$$
\begin{array}{l}
depart(ll : LIFT, dd : DIR) = \\
\quad |\ ll \notin moving \wedge (floor.ll, dd) \in \mathsf{dom}.MOVE\ |; \\
\qquad dir := dir \mathbin{\lhd\!\!\!+} \{ll \mapsto dd\}\ \| \\
\qquad moving := moving \cup \{ll\};
\end{array}
$$

end

class	action	explanation
(RND)	*reqLift*	caused by human lift user
(RND)	*reqFloor*	caused by human lift user
(DEP)	*move*	consequence of physical lift movements
(DEP)	*stop*	spatial restriction on lift movements
(IND)	*stop*	decision to stop any number of moving lifts
(IND)	*depart*	decision to commence moving stationary lifts

Table 7.1: Actions of machine *SYS_LIFT*

7.2 From Machines to Systems

Note that machine *SYS_LIFT* is internally deterministic. The nondeterminism present is purely external. No particular order for the occurrence of actions *reqLift*, *reqFloor*, *move*, *bang*, *stop*, and *depart* of machine *SYS_LIFT* is given. It is assumed that it is the responsibility of some environment of the machine to resolve the external nondeterminism. We specify such an environment and do performance analysis on the complete system consisting of controller and environment.

We distinguish three different classes of actions. This allows us to follow a structured approach in the presentation of the material. We distinguish

- actions that occur randomly, (RND)
- actions that occur dependently, (DEP)
- actions that occur independently. (IND)

Actions in class (RND) are controlled by some stochastic source. We make no further assumptions about actions in this class. Actions belonging to class (DEP) describe properties of observable quantities, the values of which change deterministically. Actions in class (IND) are controllable. For actions in this class we seek a refinement in the form of a control law. Classifications like this one are customary in control theory [30, 41, 93]. Table 7.1 presents a classification of all actions of machine *SYS_LIFT*. The first column states the class of an action, the second its name. The third column gives a short explanation to justify the presence of an action in the particular class. Actions *reqLift* and *reqFloor* model input from human users of the system. Action *move* models physical lift movements. Action *stop* is present in two classes because it can represent a decision to stop a lift, or the necessity to stop a lift because it has reached the bottom or top floor. Finally action *depart* represents a decision to commence moving.

We combine the actions of machine *SYS_LIFT* into more complex programs in a system *PAR_LIFT* to account for the parallelism present in the complete system. Each class of machine actions has a corresponding

way of combining actions belonging to it. They have in common that they all require interleaving which we present in the next section. The distinguishing features are treated in subsequent sections, where we derive system *PAR_LIFT*, the model of the actual control system we are interested in. In sections 7.2.2, 7.2.4 and 7.2.3 the programs section of system *PAR_LIFT* are derived. Finally, section 7.2.5 presents the sole action of system *PAR_LIFT*.

7.2.1 Interleaving

We say probabilistic state relations P_1 and P_2 are *non-interfering* if they commute with respect to sequential composition,

$$P_1; P_2 \;=\; P_2; P_1 \;.$$

For non-interfering probabilistic state relations P_i, where $i \in I$, we define finite interleaving as follows:

$$|||\, i : I \bullet P_i \;\;\widehat{=}\;\; |\, I = \varnothing \,| \sqcup (\textstyle\bigsqcup i : I \bullet (|||\, j : I \setminus \{i\} \bullet P_j); P_i) \;. \qquad (7.1)$$

Because of the noninterference of the P_i in (7.1) finite interleaving is equivalent to any sequential composition of the P_i. This means interleaving simply allows us to reorder the probabilistic state relations as is most convenient. If $I = \varnothing$, then

$$|||\, i : I \bullet P_i \;=\; \mathsf{skip} \;.$$

If $I = \{i_1, i_2, \ldots, i_n\}$, $n \geq 1$, then

$$|||\, i : I \bullet P_i \;=\; P_{i_1}; P_{i_2}; \ldots; P_{i_n} \;.$$

We also define binary interleaving: $(P_1 \;|||\; P_2) \;\widehat{=}\; (P_1; P_2) \sqcup (P_2; P_1)$. To make use of interleaving we have to show that the constituent probabilistic state relations P_i are non-interfering. Subsequently we can replace interleaving by sequential composition.

7.2.2 Random Actions

Because of laws (L16) and (L17) we do not formulate the following definition for nondeterministic actions. An action $\phi \in \mathcal{D}(\Gamma)$ in class (RND) occurs with a certain (constant) probability c. The probability that nothing happens is $1 - c$. More generally, we define probabilistic option for probabilistic state functions $M \in \mathcal{M}(\Gamma)$ by

$$p \;?\; M \;\;\widehat{=}\;\; M \;{}_{p}\!\oplus\; \mathsf{skip} \;.$$

We note that if state relations M_1 and M_2 are non-interfering, so are $c_1 \;?\; M_1$ and $c_2 \;?\; M_2$. The interleaving of the latter two probabilistic state relations,

$$c_1 \;?\; M_1 \;|||\; c_2 \;?\; M_2 \;,$$

models independent occurrences of M_1 and M_2 with probabilities c_1 and c_2 respectively. The joint probability of $M_1 \mid\mid\mid M_2$ is $c_1 * c_2$, and the joint probability of skip is $(1-c_1)*(1-c_2)$, and so on. Proposition 7.1 generalises this. We would fail to prove the proposition if we allowed state relations instead of state functions ϕ.

Proposition 7.1 Let $c_i \in (0,1)$ for all $i \in I$, and all M_i non-interfering:

$$\begin{aligned} &\mid\mid\mid i : I \bullet (c_i \; ? \; M_i) \;\; = \\ &\quad \bigoplus J : \mathbb{P}\, I \mid (\textstyle\prod i : J \bullet c_i) * (\textstyle\prod i : I \setminus J \bullet 1 - c_i) \bullet (\mid\mid\mid j : J \bullet M_j) \; . \end{aligned}$$

Suppose that during one unit of time a request for a lift at floor ff takes place with probability 0.05. This probability and its opposite are wrapped into set REQ to simplify the presentation of the product in proposition 7.1,

$$REQ \;\; = \;\; \{\mathsf{true} \mapsto 0.05, \mathsf{false} \mapsto 0.95\} \; .$$

For $J \subseteq I$ we have

$$(\textstyle\prod i : J \bullet 0.05) * (\textstyle\prod i : I \setminus J \bullet 0.95) \;\; = \;\; \textstyle\prod i : I \bullet REQ.(i \in J) \; .$$

The actions $SYS_LIFT.reqLift(\mathit{ff})$ are non-interfering, hence,

$$\begin{aligned} &PAR_LIFT.reqLift \\ &\quad \widehat{=} \;\; \mid\mid\mid \mathit{ff} : FLOOR \setminus floor[LIFT \setminus moving] \bullet \\ &\qquad\quad 0.05 \; ? \; SYS_LIFT.reqLift(\mathit{ff}); \\ &\quad = \;\; \bigoplus FF : \mathbb{P}(FLOOR \setminus floor[LIFT \setminus moving]) \\ &\qquad\quad \mid (\textstyle\prod \mathit{ff} : FLOOR \setminus floor[LIFT \setminus moving] \bullet REQ.(\mathit{ff} \in FF)) \bullet \\ &\qquad\qquad \mid\mid\mid \mathit{ff} : FF \bullet req := req \cup \{\mathit{ff}\}; \\ &\quad = \;\; \bigoplus FF : \mathbb{P}(FLOOR \setminus floor[LIFT \setminus moving]) \\ &\qquad\quad \mid (\textstyle\prod \mathit{ff} : FLOOR \setminus floor[LIFT \setminus moving] \bullet REQ.(\mathit{ff} \in FF)) \bullet \\ &\qquad\qquad req := req \cup FF; \end{aligned}$$

Remember that PAR_LIFT and SYS_LIFT use different models of time. Execution of machine action $SYS_LIFT.reqLift(\mathit{ff})$ is not associated with a duration. As a component of a probabilistic action system, on the other hand, program $PAR_LIFT.reqLift$ is executed within one unit of time. This means in probabilistic action system PAR_LIFT groups of machine actions $SYS_LIFT.reqLift(\mathit{ff})$ are executed within one time unit.

Suppose that a request for some lift ll to travel to another floor occurs with probability 0.1. Only one destination floor is chosen per lift, and all floors $\mathit{ff} \neq floor(ll)$ are chosen with equal probability

$$1.0/(FLOORS - 1) \; .$$

Remember that $SYS_LIFT.reqFloor(ll, ff)$ blocks execution if $ff = floor(ll)$. Similar to function REQ above we define set $ENTER$ for use in program $PAR_LIFT.reqFloor$:

$$ENTER \;=\; \{\mathsf{true} \mapsto 0.1, \mathsf{false} \mapsto 0.9\} \;.$$

Again, the actions $SYS_LIFT.reqFloor(ll, ff)$ are non-interfering.

$$
\begin{array}{ll}
\multicolumn{2}{l}{PAR_LIFT.reqFloor} \\
\mathrel{\widehat=} & |||\, ll : LIFT \setminus moving \bullet \\
& \quad 0.1\ ? \\
& \quad \bigoplus ff : FLOOR \setminus floor(ll) \mid 1.0/(FLOORS - 1) \bullet \\
& \qquad SYS_LIFT.reqFloor(ll, ff); \\
= & \bigoplus LL : \mathbb{P}(LIFT \setminus moving) \\
& \quad \mid (\prod ll : LIFT \setminus moving \bullet ENTER.(ll \in LL)) \bullet \\
& \qquad |||\, ll : LL \bullet \\
& \qquad\quad \bigoplus ff : FLOOR \setminus floor.ll \mid 1.0/(FLOORS - 1) \bullet \\
& \qquad\qquad out := out \cup \{ll \mapsto ff\}; \\
= & \bigoplus LL : \mathbb{P}(LIFT \setminus moving) \\
& \quad \mid (\prod ll : LIFT \setminus moving \bullet ENTER.(ll \in LL)) \bullet \\
& \qquad \bigoplus RR : \{RR : LL \to FLOOR \mid (\forall ll : LL \bullet RR.ll \neq floor.ll)\} \\
& \qquad\quad \mid (\prod ll : LL \bullet 1.0/(FLOORS - 1)) \bullet \\
& \qquad\qquad |||\, ll : LL \bullet out := out \cup \{ll \mapsto RR.ll\}; \\
= & \bigoplus LL : \mathbb{P}(LIFT \setminus moving) \\
& \quad \mid (\prod ll : LIFT \setminus moving \bullet ENTER.(ll \in LL)) \bullet \\
& \qquad \bigoplus RR : \{RR : LL \to FLOOR \mid RR \cap floor = \varnothing\} \\
& \qquad\quad \mid (\prod ll : LL \bullet 1.0/(FLOORS - 1)) \bullet \\
& \qquad\qquad |||\, ll : LL \bullet out := out \cup \{ll \mapsto RR.ll\}; \\
= & \bigoplus LL : \mathbb{P}(LIFT \setminus moving) \\
& \quad \mid (\prod ll : LIFT \setminus moving \bullet ENTER.(ll \in LL)) \bullet \\
& \qquad \bigoplus RR : \{RR : LL \to FLOOR \mid RR \cap floor = \varnothing\} \\
& \qquad\quad \mid (\prod ll : LL \bullet 1.0/(FLOORS - 1)) \bullet \\
& \qquad\qquad out := out \cup RR;
\end{array}
$$

7.2.3 Dependent Actions

Actions in class (DEP) are combined solely by interleaving. For $kk \neq ll$ we have $move(kk); move(ll) = move(ll); move(kk)$, hence

$$PAR_LIFT.move$$

$$
\begin{aligned}
&\mathrel{\widehat{=}} \ ||| \ ll : moving \bullet SYS_LIFT.move(ll); \\
&= \ ||| \ ll : moving \bullet \\
&\qquad | \ (floor.ll, dir.ll) \in \mathsf{dom}.MOVE \ |; \\
&\qquad floor := floor \mathbin{\lhd\!\!\!+} \{ll \mapsto MOVE(floor.ll, dir.ll)\}; \\
&= \ | \ \forall \, ll : moving \bullet (floor.ll, dir.ll) \in \mathsf{dom}.MOVE \ |; \\
&\quad floor := floor \mathbin{\lhd\!\!\!+} (\lambda \, ll : moving \bullet (MOVE.(floor.ll, dir.ll)));
\end{aligned}
$$

When a lift reaches the bottom or top floor it must stop. This is modelled by program $PAR_LIFT.bang$ utilising action $SYS_LIFT.stop(ll)$. In program $PAR_LIFT.bang$ we make use of the let statement, defined by,

$$
\mathsf{LET} \ x = E \ \mathsf{IN} \ P \ \ \widehat{=} \ \ \bigsqcup x : \{E\} \bullet P \ .
$$

The actions $SYS_LIFT.stop(ll)$ are also non-interfering. Hence,

$$
\begin{aligned}
&PAR_LIFT.bang \\
&\mathrel{\widehat{=}} \ ||| \ ll : (moving \cap floor^{\sim}[\{1\}] \cap dir^{\sim}[\{DN\}]) \cup \\
&\qquad\qquad (moving \cap floor^{\sim}[\{FLOORS\}] \cap dir^{\sim}[\{UP\}]) \bullet \\
&\qquad SYS_LIFT.stop(ll); \\
&= \ ||| \ ll : (moving \cap floor^{\sim}[\{1\}] \cap dir^{\sim}[\{DN\}]) \bullet \\
&\qquad moving := moving \setminus \{ll\} \ \| \\
&\qquad out := out \setminus \{ll \mapsto floor.ll\} \ \| \\
&\qquad req := req \setminus \{floor.ll\} \\
&\quad ||| \\
&\quad ||| \ ll : (moving \cap floor^{\sim}[\{FLOORS\}] \cap dir^{\sim}[\{UP\}]) \bullet \\
&\qquad moving := moving \setminus \{ll\} \ \| \\
&\qquad out := out \setminus \{ll \mapsto floor.ll\} \ \| \\
&\qquad req := req \setminus \{floor.ll\}; \\
&= \ \mathsf{LET} \ FD = moving \cap floor^{\sim}[\{1\}] \cap dir^{\sim}[\{DN\}] \ \mathsf{IN} \\
&\qquad moving := moving \setminus FD \ \| \\
&\qquad out := out \setminus (FD \lhd floor) \ \| \\
&\qquad req := req \setminus floor[FD]; \\
&\quad \mathsf{LET} \ FU = moving \cap floor^{\sim}[\{FLOORS\}] \cap dir^{\sim}[\{UP\}] \ \mathsf{IN} \\
&\qquad moving := moving \setminus FU \ \| \\
&\qquad out := out \setminus (FU \lhd floor) \ \| \\
&\qquad req := req \setminus floor[FU];
\end{aligned}
$$

7.2.4 Independent Actions

The program $\mathsf{skip} \sqcup R$ models a decision to do either R or nothing. The interleaving of several such programs models multiple independent decisions.

Proposition 7.2 treats this situation similar to the probabilistic one in proposition 7.1.

Proposition 7.2 Let all $R_i \in \mathcal{R}(\Gamma)$ be non-interfering. Then

$$|||\, i : I \bullet (\mathsf{skip} \sqcup R_i) \;=\; \bigsqcup J : \mathbb{P}\, I \bullet (|||\, j : J \bullet R_j)\ .$$

The controller can decide to stop lift ll if it is moving. The actions $SYS_LIFT.stop(ll)$ are non-interfering.

$$\begin{array}{ll}
\multicolumn{2}{l}{PAR_LIFT.stop} \\
\;\widehat{=} & |||\, ll : moving \bullet (\mathsf{skip} \sqcup SYS_LIFT.stop(ll)); \\
= & \bigsqcup LL : \mathbb{P}(moving) \bullet (|||\, ll : LL \bullet SYS_LIFT.stop(ll)); \\
= & \bigsqcup LL : \mathbb{P}(moving) \bullet \\
& \quad |||\, ll : LL \bullet \\
& \qquad moving := moving \setminus \{ll\} \parallel \\
& \qquad out := out \setminus \{ll \mapsto floor.ll\} \parallel \\
& \qquad req := req \setminus \{floor.ll\}; \\
= & \bigsqcup LL : \mathbb{P}(moving) \bullet \\
& \quad moving, out, req := \\
& \qquad moving \setminus LL, out \setminus (LL \lhd floor), req \setminus floor[LL];
\end{array}$$

If lift ll is not moving the controller can decide to commence moving that lift. The chosen direction dd must be UP if the lift is in the ground floor, and DN if it is in the top floor. As before the actions $SYS_LIFT.depart(ll, dd)$ are non-interfering.

$$\begin{array}{ll}
\multicolumn{2}{l}{PAR_LIFT.depart} \\
\;\widehat{=} & |||\, ll : LIFT \setminus moving \bullet \\
& \quad \mathsf{skip} \sqcup \\
& \quad \bigsqcup dd : DIR \bullet SYS_LIFT.depart(ll, dd); \\
= & \bigsqcup LL : \mathbb{P}(LIFT \setminus moving) \bullet \\
& \quad |||\, ll : LL \bullet \\
& \qquad \bigsqcup dd : DIR \bullet SYS_LIFT.depart(ll, dd); \\
= & \bigsqcup LL : \mathbb{P}(LIFT \setminus moving) \bullet \\
& \quad \bigsqcup DD : LL \to DIR \bullet \\
& \qquad |||\, ll : LL \bullet SYS_LIFT.depart(ll, DD.ll); \\
= & \bigsqcup LL : \mathbb{P}(LIFT \setminus moving) \bullet \\
& \quad \bigsqcup DD : LL \to DIR \bullet \\
& \qquad |||\, ll : LL \bullet
\end{array}$$

$$
\begin{aligned}
&\qquad\qquad | \; (floor.ll, DD.ll) \in \mathsf{dom}.MOVE \; |; \\
&\qquad\qquad\quad moving := moving \cup \{ll\} \; \| \\
&\qquad\qquad\quad dir := dir \lhd\!\!\!+ \{ll \mapsto DD.ll\}; \\
= \; & \textstyle\bigsqcup LL : \mathbb{P}(LIFT \setminus moving) \bullet \\
&\quad \textstyle\bigsqcup DD : LL \to DIR \bullet \\
&\qquad | \; \forall\, ll : LL \bullet (floor.ll, DD.ll) \in \mathsf{dom}.MOVE \; |; \\
&\qquad moving, dir := moving \cup LL, dir \lhd\!\!\!+ DD;
\end{aligned}
$$

7.2.5 The Lift System

In this section we present the actions section of system PAR_LIFT.

system PAR_LIFT
constants $FLOORS$; $LIFTS$;
...

The variables section and initialisation section are identical to the corresponding sections of machine SYS_LIFT. The components of the programs section have been introduced in sections 7.2.2 to 7.2.4.

System PAR_LIFT has one action called *cycle* defining the control cycle of the lift system. The leading test in action *cycle* sets up a cost structure for the lift system. Costs of one unit are incurred for each unanswered request and for each moving lift. Hence the associated optimisation problem is to minimise overall waiting time (measured by unanswered requests) and energy consumption (measured by the number moving lifts). The control cycle has four phases. First newly arrived requests are recorded. Afterwards the lift movements are calculated. In the third phase it is decided which lifts to stop. Finally it is decided which lifts should depart, i.e. move in the next cycle.

actions
  $cycle \;=$
    $| \; \mathsf{card}.req + \mathsf{card}.out + \mathsf{card}.moving \; |;$
    $reqLift; reqFloor; move; bang; stop; depart;$
end

Unfortunately system PAR_LIFT can not be analysed with our software tool because programs *stop* and *depart* contain nondeterminism (see section 6.2.2). In the next section we introduce a transformation that we use in section 7.4 to split action *cycle* into two separate actions. The resulting system can be analysed with the software tool.

7.3 Time Scale Transformation

Let **A** and **B** be two live probabilistic action systems such that for all $v \in \mathsf{seq}_\infty \mathbb{R}_{\geq 0}$:

$$\mathsf{itr}.\mathbf{A}.v \;\;\Leftrightarrow\;\; \mathsf{itr}.\mathbf{B}.(\sigma.v) \ ,$$

where $\sigma : \mathsf{seq}_\infty \mathbb{R}_{\geq 0} \rightarrow \mathsf{seq}_\infty \mathbb{R}_{\geq 0}$ is defined by $\sigma.v.(2i-1) \mathrel{\widehat{=}} v.i$ and $\sigma.v.(2i) \mathrel{\widehat{=}} 0$ for all $i \in \mathbb{N}_1$.

From proposition 8.1.1 in [98] we know that if the policy underlying trace v is stationary, then

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} v.i \;\; = \;\; \limsup_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} v.i \ . \tag{7.2}$$

Using this we can relate the optimal policies of systems **A** and **B** by the following proposition.

Proposition 7.3 Let $v \in \mathsf{itr}.\mathbf{A}$ be an infinite trace which belongs to a stationary policy, and $\sigma.v \in \mathsf{itr}.\mathbf{B}$. Then the following equality holds:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} v.i \;\; = \;\; \lim_{n\to\infty} \frac{2}{n} \sum_{i=1}^{n} \sigma.v.i \ . \tag{7.3}$$

Equation (7.2) implies that the traces of the two systems **A** and **B** belonging to stationary policies are related by (7.3). This includes the optimal traces. Assume the optimal value of system **A** is $2o$ belonging to infinite trace w. Then the optimal value of system **B** is o attained with trace $\sigma.w$.

Proposition 7.4 If **A** and **B** (as above) have deterministic initialisations, then

$$\mathsf{val}.\mathbf{A} \;\; = \;\; 2 * \mathsf{val}.\mathbf{B} \ .$$

7.4 More Actions

The lift system of section 7.2.5 is of no practical use to us since we can not analyse it automatically. In this section we present a variant of that system which we can analyse with our tool. The idea we follow is simple. We split action *cycle* into two actions *request* and *serve* which are executed alternately. Figure 7.1 shows the transformed system *PER_LIFT*.

A new set *TURN* consisting of two values *REQUEST* and *SERVE*, and a variable *turn* of type *TURN* are introduced. If *turn* has value *REQUEST*, action *request* is executed, otherwise action *serve*. Initially *turn* is set to *REQUEST*. The decision to split action *PAR_LIFT.cycle* this way is arbitrary. An alternative would have been to detach card.*moving* from the initial

cost statement attach it to $PER_LIFT.serve$, leaving $\mathsf{card}.req + \mathsf{card}.out$ in $PER_LIFT.request$. We intend to use the transformation from section 7.3. So the latter approach, where *serve* has a non-zero expected cost, is not applicable. We have chosen the approach of figure 7.1 because it corresponds to a simple transformation which is easy to prove.

```
system PER_LIFT
constants FLOORS;  LIFTS;
sets
   ...
   TURN  =  REQUEST | SERVE;
variables
   ...
   turn  :  TURN;
initialisation
   ... ||
   turn := REQUEST;
programs
   ...
actions
   request  =
      | turn = REQUEST |; turn := SERVE;
      | card.req + card.out + card.moving |;
      reqLift; reqFloor;
   serve  =
      | turn = SERVE |; turn := REQUEST;
      move; bang; stop; depart;
end
```

Figure 7.1: Time-wise transformed lift

An execution of action *request* incurs a cost of $\mathsf{card}.req + \mathsf{card}.out + \mathsf{card}.moving$, and an execution of action *serve* a cost of zero. In our discrete model of time, one unit of time in system PAR_LIFT corresponds to two units of time in system PER_LIFT. We have

$$| \; turn = REQUEST \; |; PAR_LIFT.cycle$$
$$= \; PER_LIFT.request; PER_LIFT.serve \; .$$

Hence, system PER_LIFT incurs the same costs in two units of time as PAR_LIFT does in one unit. Based on section 7.3 we show that both systems have in essence the same optimal implementation:

Let *optServe* be the optimal implementation of *serve* with the leading part $| \; turn = SERVE \; |; turn := REQUEST$ removed. Then action *optCycle*

given by

$$\begin{aligned} &optCycle \ = \\ &\quad |\ \mathsf{card}.req + \mathsf{card}.out + \mathsf{card}.moving\ |; \\ &\quad reqLift; reqFloor; \\ &\quad optServe; \end{aligned}$$

is an optimal implementation of action *cycle* because, by proposition 7.4, its optimal value is $2 * \mathsf{val}.PER_LIFT$. This value is attained for action *optCycle*. Hence *optCycle* is optimal.

7.5 Fewer States

System *PER_LIFT* in the last section has a feasible number of transitions per state but is has also many more states than the original system *PAR_LIFT*. In this section we present two fusions to reduce the number of states. Remember that a fusion is a deterministic simulation (see chapter 4). The resulting system *RED_LIFT*, incorporating the two fusions, is finally used in section 7.6 as input to our software tool.

7.5.1 State Aggregation

There is more than one way to aggregate states. It is worth varying the approach to find a good fusion. Fusion *arrange* presented below arranges lifts in ascending order with respect to their location. Alternatively one might order the lifts so that, say, moving lifts always have numbers $1 \mathinner{..} n$ where $n = \mathsf{card}.moving$. Subsequently, moving and stationary lifts might be ordered separately in ascending order. We do not follow that approach but remark that it is much less effective than the one presented. In particular, the advantage of fusion *arrange* is that it takes variable *out* into account. As the considered system gets larger the effect of *out* on the number of reachable states increases. The opposite is true for variable *moving*.

We distinguish two kinds of aggregation. First we eliminate redundant states. If in some state the value of some variable has no significance, then we can arbitrarily choose a value for that variable. The second kind is based on the observation that a system may contain symmetries in its behaviour. Redundant states give rise to such symmetries. Still we distinguish the two kinds because the first one yields very simple fusions.

Redundant States

We observe that once a lift is stopped the directional information associated with it becomes irrelevant. At initialisation time all lifts are stopped and set

to face upwards. So, we decide to let stopped lifts always to face upwards. This is what fusion *faceUp* does.

$$\begin{array}{l} faceUp \;=\; \\ \quad dir := dir \mathbin{\lhd\!\!+} (LIFT \setminus moving) \times \{UP\}; \end{array}$$

We have to prove idempotency of program *faceUp*,

$$faceUp; faceUp \;=\; faceUp \;. \tag{7.4}$$

Since the expression $(LIFT \setminus moving) \times \{UP\}$ does not contain a reference to variable *dir*, equation 7.4 is satisfied. Next we have to show that (FUS) is satisfied by *faceUp*, *request* and *serve*:

$$request; faceUp \;=\; faceUp; request; faceUp \;, \tag{7.5}$$

$$serve; faceUp \;=\; faceUp; serve; faceUp \;. \tag{7.6}$$

Action *request* does not refer to variable *dir*, hence,

$$request; faceUp \;=\; faceUp; request \;,$$

and equation 7.5 is satisfied. Programs *move*, *bang* and *depart* in action *serve* refer to *dir*. We prove

$$move; faceUp \;=\; faceUp; move; faceUp \;, \tag{7.7}$$

$$bang; faceUp \;=\; faceUp; bang; faceUp \;, \tag{7.8}$$

$$depart; faceUp \;=\; faceUp; depart; faceUp \;. \tag{7.9}$$

The rest follows by proposition 4.11. Equations 7.7 and 7.8 are satisfied because *move* only refers to $moving \lhd dir$ which is not changed by program *faceUp*. The same is true for program *bang* because

$$moving \cap dir^{\sim}[\{dd\}] \;=\; (moving \lhd dir)^{\sim}[\{dd\}]$$

for $dd \in DIR$. With regard to program *depart* it is sufficient to consider the fragment

$$departFrag \;\widehat{=}\; moving, dir := moving \cup LL, dir \mathbin{\lhd\!\!+} DD$$

where $LL \subseteq LIFT \setminus moving$ and $DD \in LL \rightarrow DIR$. Using substitution we calculate:

$$\begin{array}{l} [departFrag]\,(dir \mathbin{\lhd\!\!+} (LIFT \setminus moving) \times \{UP\}) \\ \quad = \; (dir \mathbin{\lhd\!\!+} DD) \mathbin{\lhd\!\!+} (LIFT \setminus (moving \cup LL)) \times \{UP\} \\ \quad = \; dir \mathbin{\lhd\!\!+} (DD \cup (LIFT \setminus (moving \cup LL)) \times \{UP\}) \\ \quad = \; (dir \mathbin{\lhd\!\!+} (LIFT \setminus (moving \cup LL)) \times \{UP\}) \mathbin{\lhd\!\!+} DD \;. \end{array}$$

It follows:

$$\begin{aligned}
& departFrag; faceUp \\
= \;& dir := (dir \lhd\!\!\!+ (LIFT \setminus (moving \cup LL)) \times \{UP\}) \lhd\!\!\!+ DD; \\
& moving, dir := moving \cup LL, dir \lhd\!\!\!+ DD; \\
& dir := dir \lhd\!\!\!+ (LIFT \setminus moving) \times \{UP\} \\
= \;& dir := dir \lhd\!\!\!+ (LIFT \setminus (moving \cup LL)) \times \{UP\}; \\
& departFrag; faceUp \\
= \;& dir := dir \lhd\!\!\!+ (LIFT \setminus moving) \times \{UP\}; \\
& departFrag; faceUp \\
= \;& faceUp; departFrag; faceUp \ .
\end{aligned}$$

This implies 7.9.

Variable Ordering

It is only relevant how many lifts are at which level. We can discard the information about which particular lift is at some floor. We rearrange the lifts in ascending order. Lifts on the same floor are ordered according to the unmet requests in variable *out*.

Let $GL \in LIFT \to floor[LIFT]$ be a function that assigns lifts to floors. The two predicates *ascend* and *invar* restrict the possible values of GL. If GL satisfies *ascend*, then all lifts are assigned floors in ascending order:

$$ascend \ \widehat{=} \ \forall\, xx : LIFT,\ yy : LIFT \bullet xx < yy \Rightarrow GL.xx \leq GL.yy \ .$$

If GL satisfies *invar*, then the number of lifts at each floor remains invariant:

$$invar \ \widehat{=} \ \forall\, xx : floor[LIFT] \bullet \mathsf{card}.floor^{\sim}[\{xx\}] = \mathsf{card}.GL^{\sim}[\{xx\}] \ .$$

Function GL is not uniquely determined by *ascend* and *invar*. However it becomes unique as a consequence of the lift reordering, $PP \in LIFT \to LIFT$, presented next. Let PP satisfy predicate *bridge*. Then the reordering achieves an exact match between the locations of the lifts stated in variable *floor*, and those stated in function GL:

$$bridge \ \widehat{=} \ \forall\, xx : LIFT \bullet floor.(PP.xx) = GL.xx \ .$$

If two lifts are on the same floor according to the reordering PP, then a further ordering based on variable *out* is used. For $xx \in LIFT$ the set $out[\{xx\}]$ contains all floors which lift xx is expected to visit. We define a total order SMU on $\mathbb{P}(FLOOR)$ to achieve a reordering of variable *out* that respects GL:

$$SMU \ = \ \{xx : \mathbb{P}(FLOOR),\ yy : \mathbb{P}(FLOOR) \mid$$

$$
\begin{aligned}
&\quad \mathsf{card}.xx \leq \mathsf{card}.yy \wedge \\
&\quad (\mathsf{card}.xx = \mathsf{card}.yy \wedge xx \neq yy \Rightarrow \mathsf{min}.(xx \setminus yy) \leq \mathsf{min}.(yy \setminus xx)) \\
&\};
\end{aligned}
$$

If PP satisfies predicate *locate*, then it also orders variables *out*, *dir*, *moving* in ascending order. Lifts on the same floor, $GL.xx = GL.yy$, are ordered with respect to SMU. If their requests are also identical, $out[\{PP.xx\}] = out[\{PP.yy\}]$, then they are ordered so that moving lifts come last. If two lifts are either both moving or stationary, $PP.xx \in moving \Leftrightarrow PP.yy \in moving$, then the lifts are ordered so that lifts going down come first, and then lifts going up. If the lifts also face the same direction, $dir.(PP.xx) = dir.(PP.yy)$, then both lifts are in the same state. In this case their existing order $PP.xx < PP.yy$ is used.

$$
\begin{aligned}
locate \;\hat{=}\; & \forall\, xx : LIFT,\ yy : LIFT \bullet \\
& \quad GL.xx = GL.yy \wedge xx < yy \Rightarrow \\
& \quad\quad out[\{PP.xx\}] \mapsto out[\{PP.yy\}] \in SMU \wedge \\
& \quad\quad out[\{PP.xx\}] = out[\{PP.yy\}] \Rightarrow \\
& \quad\quad\quad (PP.xx \in moving \Rightarrow PP.yy \in moving) \wedge \\
& \quad\quad\quad (PP.xx \in moving \Leftrightarrow PP.yy \in moving) \Rightarrow \\
& \quad\quad\quad\quad (dir.(PP.xx) = UP \Rightarrow dir.(PP.yy) = UP) \wedge \\
& \quad\quad\quad\quad dir.(PP.xx) = dir.(PP.yy) \Rightarrow \\
& \quad\quad\quad\quad\quad PP.xx < PP.yy\ .
\end{aligned}
$$

Predicate *locate* fixes the value of PP and as consequence that of GL. Now we define fusion *arrange*:

$$
\begin{aligned}
& arrange \;=\; \\
& \quad \bigsqcup GL : LIFT \rightarrow floor[LIFT] \bullet \mid ascend \wedge invar \mid; \\
& \quad\quad \bigsqcup PP : LIFT \rightarrow LIFT \bullet \mid bridge \wedge locate \mid; \\
& \quad\quad\quad floor := GL \parallel \\
& \quad\quad\quad out := \{ll : LIFT,\ ff : FLOOR \mid PP.ll \mapsto ff \in out\} \parallel \\
& \quad\quad\quad dir := \{ll : LIFT,\ dd : DIR \mid PP.ll \in dd \in dir\} \parallel \\
& \quad\quad\quad moving := \{ll : LIFT \mid PP.ll \in moving\};
\end{aligned}
$$

Program *arrange* is nonempty and deterministic because for any choice of *floor* and *out* there are unique functions GL and PP satisfying

$$
fix \;\hat{=}\; ascend \wedge invar \wedge bridge \wedge locate\ .
$$

Idempotency of *arrange* follows from the uniqueness of GL and PP, and because of the strict monotonicity of PP when two lifts xx and yy are in the

same state:

$$\begin{aligned} equal.(xx, yy) \;\widehat{=}\; & floor.xx = floor.yy \wedge \\ & out[\{xx\}] = out[\{yy\}] \wedge \\ & (xx \in moving \Leftrightarrow yy \in moving) \wedge \\ & dir.xx = dir.yy \;. \end{aligned}$$

We show that programs *arrange*, *request* and *serve* satisfy (FUS):

$$request; arrange \;=\; arrange; request; arrange \;, \tag{7.10}$$

$$serve; arrange \;=\; arrange; serve; arrange \;. \tag{7.11}$$

By proposition 4.11 it is sufficient to prove the corresponding properties for all of the programs, *reqLift*, *reqFloor*, *move*, *bang*, *stop*, *depart*, and the test $|\,\mathsf{card}.req + \mathsf{card}.out + \mathsf{card}.moving\,|$. Equations 7.10 and 7.11 follow. The test and *arrange* are non-interfering because *PP* is a permutation. Programs *reqLift* and *arrange* have no variables in common. So they are non-interfering. The truth of $reqFloor; arrange = arrange; reqFloor; arrange$ relies on the fact that predicate *fix* orders lifts *xx* and *yy* up to equality, $equal.(xx, yy)$, in which case *xx* and *yy* are indistinguishable. The same argument applies to programs *move*, *bang*, *stop*, and *depart*.

7.5.2 The Reduced Lift System

We combine the two fusions from section 7.5.1 into a single fusion.

$$reduce \;\widehat{=}\; faceUp; arrange;$$

The two fusions *faceUp* and *arrange* are non-interfering. Thus, by proposition 4.10, program *reduce* is a fusion of *PER_LIFT*.

Note that the two fusions *faceUp* and *arrange* contain some information about the optimal implementation. From fusion *faceUp* we see that directional information on stationary lifts is irrelevant. Fusion *arrange* implies that the identity of a lift is irrelevant. Only their location is important.

The reduced lift system *RED_LIFT* is derived from system *PER_LIFT*. Most of system *PER_LIFT* is adopted unchanged:

```
system RED_LIFT
...
programs
  ...
  reduce = faceUp; arrange
actions
  request = PER_LIFT.request; reduce
  serve = PER_LIFT.serve; reduce
end
```

system	FLOORS	states	transitions
PER_LIFT	2	368	1016
RED_LIFT	2	144	368
PER_LIFT	3	12800	54656
RED_LIFT	3	4728	20064
RED_LIFT	4	99200	516352

Table 7.2: Sizes of systems *PER_LIFT* and *RED_LIFT*

We can use the initialisation of *PER_LIFT* unchanged because

$$PER_LIFT.\mathsf{initialisation}; reduce \quad = \quad PER_LIFT.\mathsf{initialisation} \ .$$

7.6 Evaluation and Discussion

We have analysed systems *PER_LIFT* and *RED_LIFT* with $LIFTS = 2$ and varying values of *FLOORS* (see table 7.2). The optimal solution of the system with four floors produced by the software tool is a table similar to the one in section 6.4. However, it has 57633 rows. We have written a small tool that stores this table in the database so that it can be queried conveniently. Still it is extremely difficult to gain insight into the optimal solution. Using the table directly seems impractical for larger systems. Instead, for systems of this size we would prefer another tool component that would produce a more compact representation. In the ideal case this would be a syntactical probabilistic action system.

The state space of system *RED_LIFT* is about two fifth of the size of the state space of system *PER_LIFT*. The number of transitions is reduced by a similar amount. We have not been able to determine the size of system *PER_LIFT* with four floors. The expansion of the corresponding *RED_LIFT* takes about 24 hours on a 500MHz computer with 256MB of RAM.

The relationship between systems *PER_LIFT* and *RED_LIFT* is not a refinement. We have used the time scale transformation to relate the inner structure of the two systems. As a result we are able to interchange them in the performance analysis. The transformation extends the refinement notion of probabilistic action systems by incorporating properties of the employed optimality criterion. It is invariant with respect to the optimal value though.

In this chapter we have refined what was external nondeterminism in the machine *SYS_LIFT*. This corresponds to the view that we have determined a controller that uses machine *SYS_LIFT* to control the operation of a lift system. We have not dealt with machines that are internally nondeterministic. This issue is left for further investigation. Using the approach of section

7.2 one would have to map some nondeterminism to probabilistic and some to nondeterministic choice.

Conclusion

We have introduced the formalism of probabilistic action systems. Their behaviour is described by traces of expected costs a system may incur during operation. Neither state nor actions are visible. This corresponds to the view that a complete system is modelled with no further possible synchronisation. We have also presented a notion of cost refinement together with simulation-based proof rules. The proof rules used with probabilistic action systems are similar to those used with standard action systems. This facilitates the use of probabilistic action systems for someone familiar with standard action systems. The same holds for the definition of refinement itself. Cost refinement resembles trace refinement.

The expected costs specified in probabilistic action systems encode performance objectives. The optimisation criterion on which the performance measure associated with a system is based is not fixed. This makes it possible to choose an appropriate criterion in different application contexts. Refinement is also not linked to a particular optimisation criterion. A refinement may lead to a non-optimal implementation. An initial probabilistic action system specifies a boundary for the best possible performance any refinement may achieve. Although one generally aims at finding an optimal implementation optimality is not necessary for the implementation to be correct. Performance analysis as used in our approach only assists in the refinement process.

Using the probabilistic program notation introduced in chapter 3 refinement proofs are mostly done by algebraic reasoning. Calculations involving probability densities on the semantical level are avoided. The algebraic laws concerning probabilistic choice (L16) to (L19) are rather restrictive limiting their use to programs that contain little nondeterminism. However, the same restriction applies to proof rules 4.4 and 4.7 where simulations must be nondeterministic. We have shown how refinement can be used to justify standard abstraction techniques used in queueing theory, e.g. replacing sets or sequences of individuals by counters (see example 4.6). Another important use of refinement is state aggregation. Because state spaces tend to be large, state aggregation is crucial in making efficient tool support available. We have used state aggregation in the example of section 4.5 and the case study of chapter 7. Subsequently we used our software tool to analyse

the system's performance and find an optimal implementation. In the case study we found that after state aggregation there were still too many states to comprehend the output presently produced by the tool.

Tool support in the use of probabilistic action systems is essential. Performing the necessary calculations by hand is impractical. The program notation and the behavioural model of probabilistic action systems have been formulated with the implementation of a software tool in mind. They both are closely related to Markov decision process used in stochastic dynamic programming. For dynamic programming to be applicable a system must be live, i.e. it must never stop. Depending on the dynamic programming algorithm used to compute an optimal implementation, further restrictions on probabilistic action systems may be required. E.g. when using value iteration we require connectedness. Liveness is checked automatically in an early phase by our tool. Connectedness can be checked automatically as well (see section 5.3).

Further Work

The lack of a tool component that translates the table of plays corresponding into an optimal implementation of a system is a serious shortcoming. As the state spaces of systems being analysed becomes larger it gets more and more difficult to comprehend the output produced by the present tool. Ideally, the output of the new component would be a probabilistic action system.

In the example of section 5.4 it was found that we would need unbounded counters to express a more appropriate performance objective. This means the system would have a countably infinite state space. In this work we have only dealt with finite state spaces. The program notation introduced in chapter 3 is not restricted to finite state spaces. This restriction is imposed when probabilistic action systems are introduced. Simply removing it the existence of an optimal implementation is not guaranteed anymore. We think this is not acceptable because a developer using probabilistic action systems should not have to prove the existence of an optimal implementation. However there are conditions under which optimal implementations exist [110]. The extension to infinite state spaces should be carefully crafted so that the simplicity of the present approach remains intact.

Though powerful enough for all applications encountered in this work proof rule 4.4 appears rather strong. More powerful rules ought to be investigated. Similar to the case of standard action systems there should be forward and backward simulations. Once these have been discovered it should be attempted to establish a completeness result as well.

Probabilistic action systems are based on a model of discrete time. Each transition takes one unit of time. One could introduce real-time where transitions have different durations. Corresponding continuous-time Markov de-

cision processes and algorithms to solve them have been investigated thoroughly [98, 110]. In the continuous-time model transition durations are distributed exponentially.

Appendix A

Proofs

Proof of law (L3)

We prove law (L3) the claim for probabilistic state relations. The case of extended probabilistic state relations is proven analogously. Let P, Q, R be probabilistic state relations, τ a state, and f a probabilistic state.

$$
\begin{array}{l}
((P;Q);R).\tau.f \\
\quad \Leftrightarrow\ \exists\, g : (P;Q).\tau,\ M : \mathsf{fun}.R \bullet \mathsf{car}.g \subseteq \mathsf{dom}.R \wedge f = g * M \\
\quad \Leftrightarrow\ \exists\, g : (P;Q).\tau,\ M : \mathsf{fun}.R \bullet \\
\qquad\qquad \exists\, h : P.\tau,\ N : \mathsf{fun}.Q \bullet \\
\qquad\qquad\quad \mathsf{car}.h \subseteq \mathsf{dom}.Q \wedge g = h * N \wedge \\
\qquad\qquad\quad \mathsf{car}.g \subseteq \mathsf{dom}.R \wedge f = g * M \\
\quad \Leftrightarrow\ \exists\, M : \mathsf{fun}.R \bullet \\
\qquad\qquad \exists\, h : P.\tau,\ N : \mathsf{fun}.Q \bullet \\
\qquad\qquad\quad \mathsf{car}.h \subseteq \mathsf{dom}.Q \wedge \\
\qquad\qquad\quad \mathsf{car}.(h * N) \subseteq \mathsf{dom}.R \wedge f = (h * N) * M \\
\quad \Leftrightarrow\ \exists\, M : \mathsf{fun}.R,\ h : P.\tau,\ N : \mathsf{fun}.Q \bullet \\
\qquad\qquad \mathsf{car}.h \subseteq \mathsf{dom}.Q \wedge \\
\qquad\qquad \mathsf{car}.(h * N) \subseteq \mathsf{dom}.R \wedge f = h * (N * M) \\
\quad \Leftrightarrow\ \exists\, M : \mathsf{fun}.R,\ h : P.\tau,\ N : \mathsf{fun}.Q \bullet \\
\qquad\qquad \mathsf{car}.h \subseteq \mathsf{dom}.Q \wedge \\
\qquad\qquad (\forall\, \tau' : \mathsf{car}.h \bullet \mathsf{car}.(N.\tau') \subseteq \mathsf{dom}.R) \wedge \\
\qquad\qquad f = h * (N * M) \\
\quad \Leftrightarrow\ \exists\, M : \mathsf{fun}.R,\ h : P.\tau,\ N : \mathsf{fun}.Q \bullet \\
\qquad\qquad (\forall\, \tau' : \mathsf{car}.h \bullet \tau' \in \mathsf{dom}.Q \wedge \mathsf{car}.(N.\tau') \subseteq \mathsf{dom}.R) \wedge \\
\qquad\qquad f = h * (N * M) \\
\quad \Leftrightarrow\ \exists\, M : \mathsf{fun}.R,\ h : P.\tau,\ N : \mathsf{fun}.Q \bullet
\end{array}
$$

$$
\begin{aligned}
&\quad (\forall\, \tau' : \mathsf{car}.h \bullet \tau' \in \mathsf{dom}.Q \wedge N.\tau' \in Q.\tau' \wedge \mathsf{car}.(N.\tau') \subseteq \mathsf{dom}.R) \wedge \\
&\quad f = h * (N * M) \\
\Leftrightarrow\ & \exists\, M : \mathsf{fun}.R,\ h : P.\tau,\ N : \mathsf{fun}.Q \bullet \\
&\quad (\forall\, \tau' : \mathsf{car}.h \bullet \tau' \in \mathsf{dom}.(Q;R)) \wedge \\
&\quad f = h * (N * M) \\
\Leftrightarrow\ & \exists\, M : \mathsf{fun}.R,\ h : P.\tau,\ N : \mathsf{fun}.Q \bullet \\
&\quad \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge \\
&\quad f = h * (N * M) \\
\Leftrightarrow\ & \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge \\
&\quad \exists\, N : \mathsf{fun}.Q,\ M : \mathsf{fun}.R \bullet \\
&\qquad f = h * (N * M) \\
\Leftrightarrow\ & \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge \\
&\quad \exists\, K \bullet f = h * K \wedge \\
&\qquad \exists\, N : \mathsf{fun}.Q,\ M : \mathsf{fun}.R \bullet \\
&\qquad\quad K = N * M \\
\Leftrightarrow\ & \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge \\
&\quad \exists\, K \bullet f = h * K \wedge \\
&\qquad \exists\, N \bullet \\
&\qquad\quad (\forall\, \tau' : \mathsf{dom}.(Q;R) \bullet N.\tau' \in Q.\tau' \wedge \mathsf{car}.(N.\tau') \subseteq \mathsf{dom}.R) \wedge \\
&\qquad\quad \exists\, M : \mathsf{fun}.R \bullet \\
&\qquad\qquad K = N * M \\
\Leftrightarrow\ & \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge \\
&\quad \exists\, K \bullet f = h * K \wedge \\
&\qquad \exists\, N \bullet \\
&\qquad\quad \forall\, \tau' : \mathsf{dom}.(Q;R) \bullet \\
&\qquad\qquad N.\tau' \in Q.\tau' \wedge \mathsf{car}.(N.\tau') \subseteq \mathsf{dom}.R \wedge \\
&\qquad\qquad \exists\, M : \mathsf{fun}.R \bullet K.\tau' = N.\tau' * M \\
\Leftrightarrow\ & \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge \\
&\quad \exists\, K \bullet f = h * K \wedge \\
&\qquad \forall\, \tau' : \mathsf{dom}.(Q;R) \bullet \\
&\qquad\quad \exists\, g : Q.\tau' \bullet \mathsf{car}.g \subseteq \mathsf{dom}.R \wedge \\
&\qquad\quad \exists\, M : \mathsf{fun}.R \bullet K.\tau' = g * M \\
\Leftrightarrow\ & \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge \\
&\quad \exists\, K \bullet f = h * K \wedge \\
&\qquad \forall\, \tau' : \mathsf{dom}.(Q;R) \bullet \\
&\qquad\quad \exists\, g : Q.\tau',\ M : \mathsf{fun}.R \bullet \mathsf{car}.g \subseteq \mathsf{dom}.R \wedge K.\tau' = g * M
\end{aligned}
$$

$$\Leftrightarrow \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge$$
$$\exists\, K \bullet f = h * K \wedge (\forall\, \tau' : \mathsf{dom}.(Q;R) \bullet K.\tau' \in (Q;R).\tau')$$
$$\Leftrightarrow \exists\, h : P.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.(Q;R) \wedge$$
$$\exists\, K : \mathsf{fun}.(Q;R) \bullet f = h * K$$
$$\Leftrightarrow (P;(Q;R)).\tau.f\ .$$

Proof of law (L9)

Let R be a state relation, P, Q probabilistic state relations, τ a state, and f a probabilistic state.

$$(R;(P \sqcup Q)).\tau.f$$
$$\Leftrightarrow \exists\, \tau' : R.\tau,\ M : \mathsf{fun}.(P \sqcup Q) \bullet f = \chi.\tau' * M$$
$$\Leftrightarrow \exists\, \tau' : R.\tau,\ M : \mathsf{fun}.(P \sqcup Q) \bullet f = M.\tau'$$
$$\Leftrightarrow \exists\, \tau' : R.\tau,\ g : (P \sqcup Q).\tau' \bullet f = g$$
$$\Leftrightarrow (\exists\, \tau' : R.\tau,\ g : P.\tau' \bullet f = g) \vee (\exists\, \tau' : R.\tau,\ g : Q.\tau' \bullet f = g)$$
$$\Leftrightarrow (R;P).\tau.f \vee (R;Q).\tau.f$$
$$\Leftrightarrow ((R;P) \sqcup (R;Q)).\tau.f\ .$$

Proof of law (L12)

Let P, Q, R be probabilistic state relations, τ a state, and f a probabilistic state.

$$P \parallel (Q \sqcup R).\tau.f$$
$$\Leftrightarrow \exists\, g : P.\tau,\ h : (Q \sqcup R).\tau \bullet f = g \parallel h$$
$$\Leftrightarrow \exists\, g : P.\tau,\ h : Q.\tau \bullet f = g \parallel h \vee \exists\, g : P.\tau,\ h : R.\tau \bullet f = g \parallel h$$
$$\Leftrightarrow (P \parallel Q).\tau.f) \vee (P \parallel R).\tau.f)$$
$$\Leftrightarrow ((P \parallel Q) \sqcup (P \parallel R)).\tau.f)\ .$$

Proof of law (L16)

Let M be a probabilistic state function, P, Q probabilistic state relations, $c \in (0,1)$ a probability, τ a state, and f a probabilistic state.

$$M;(P\ {}_c\!\oplus\ Q).\tau.f$$
$$\Leftrightarrow \exists\, g \bullet g = M.\tau \wedge$$
$$\exists\, N : \mathsf{fun}.(P\ {}_c\!\oplus\ Q) \bullet \mathsf{car}.g \subseteq \mathsf{car}.N \wedge f = g * N$$
$$\Leftrightarrow \exists\, g \bullet g = M.\tau \wedge$$
$$\exists\, N : \mathsf{fun}.P,\ K : \mathsf{fun}.Q \bullet$$
$$\mathsf{car}.g \subseteq \mathsf{car}.(c * N + (1-c) * K) \wedge$$

$$
\begin{aligned}
& \quad f = g * (c * N + (1 - c) * K) \\
\Leftrightarrow\ & \exists\, g \bullet g = M.\tau \wedge \\
& \quad \exists\, N : \mathsf{fun}.P,\ K : \mathsf{fun}.Q \bullet \\
& \quad\quad \mathsf{car}.g \subseteq \mathsf{car}.N \wedge \mathsf{car}.g \subseteq \mathsf{car}.K \wedge \\
& \quad\quad f = c * (g * N) + (1 - c) * (g * K) \\
\Leftrightarrow\ & \exists\, g \bullet g = M.\tau \wedge \\
& \quad \exists\, h,\ l \bullet \\
& \quad\quad (\exists\, N : \mathsf{fun}.P \bullet \mathsf{car}.g \subseteq \mathsf{car}.N \wedge h = g * N) \wedge \\
& \quad\quad (\exists\, K : \mathsf{fun}.Q \bullet \mathsf{car}.g \subseteq \mathsf{car}.K \wedge l = g * K) \wedge \\
& \quad\quad f = c * h + (1 - c) * l \\
\Leftrightarrow\ & \exists\, h,\ l \bullet \\
& \quad (\exists\, N : \mathsf{fun}.P,\ g \bullet g = M.\tau \wedge \mathsf{car}.g \subseteq \mathsf{car}.N \wedge h = g * N) \wedge \\
& \quad (\exists\, K : \mathsf{fun}.Q,\ g \bullet g = M.\tau \wedge \mathsf{car}.g \subseteq \mathsf{car}.K \wedge l = g * K) \wedge \\
& \quad f = c * h + (1 - c) * l \\
\Leftrightarrow\ & \exists\, h : (M; P).\tau,\ l : (M; Q).\tau \bullet f = c * h + (1 - c) * l \\
\Leftrightarrow\ & ((M; P)\ {}_c\!\oplus\ (M; Q)).\tau.f\ .
\end{aligned}
$$

Proof of law (L17)

Let M be a probabilistic state function, P, Q probabilistic state relations, $p \in \Gamma \to (0, 1)$ a probabilities, τ a state, and f a probabilistic state.

$$
\begin{aligned}
& ((P\ {}_p\!\oplus\ Q); M).\tau.f \\
& \Leftrightarrow\ \exists\, g : (P\ {}_p\!\oplus\ Q).\tau \bullet \mathsf{car}.g \subseteq \mathsf{dom}.M \wedge f = g * M \\
& \Leftrightarrow\ \exists\, g : P.\tau,\ h : Q.\tau \bullet \\
& \quad\quad \mathsf{car}.(p.\tau * g + (1 - p.\tau) * h) \subseteq \mathsf{dom}.M \wedge \\
& \quad\quad f = (p.\tau * g + (1 - p.\tau) * h) * M \\
& \Leftrightarrow\ \exists\, g : P.\tau,\ h : Q.\tau \bullet \\
& \quad\quad \mathsf{car}.g \subseteq \mathsf{dom}.M \wedge \mathsf{car}.h \subseteq \mathsf{dom}.M \wedge \\
& \quad\quad f = p.\tau * (g * M) + (1 - p.\tau) * (h * M) \\
& \Leftrightarrow\ \exists\, l,\ k \bullet \\
& \quad\quad (\exists\, g : P.\tau \bullet \mathsf{car}.g \subseteq \mathsf{dom}.M \wedge l = g * M) \wedge \\
& \quad\quad (\exists\, h : Q.\tau \bullet \mathsf{car}.h \subseteq \mathsf{dom}.M \wedge k = h * M) \wedge \\
& \quad\quad f = p.\tau * l + (1 - p.\tau) * k \\
& \Leftrightarrow\ \exists\, l : (P; M).\tau,\ k : (Q; M).\tau \bullet f = p.\tau * l + (1 - p.\tau) * k \\
& \Leftrightarrow ((P; M)\ {}_p\!\oplus\ (Q; M)).\tau.f
\end{aligned}
$$

Proof of law (L18)

Let M be a probabilistic state function, P, Q probabilistic state relations, $p : \Gamma \to (0,1)$ probabilities, τ a state, and f a probabilistic state.

$$\begin{aligned}
&(M \parallel (P \ {}_{p}\!\oplus\ Q)).\tau.f \\
&\Leftrightarrow\ \exists\, g : (P \ {}_{p}\!\oplus\ Q).\tau \bullet f = M.\tau \parallel g \\
&\Leftrightarrow\ \exists\, g : P.\tau,\ h : Q.\tau \bullet f = M.\tau \parallel (p.\tau * g + (1 - p.\tau) * h) \\
&\Leftrightarrow\ \exists\, g : P.\tau,\ h : Q.\tau \bullet f = p.\tau * (M.\tau \parallel g) + (1 - p.\tau) * (M.\tau \parallel h) \\
&\Leftrightarrow\ \exists\, g : (M \parallel P).\tau,\ h : (M \parallel Q).\tau \bullet f = p.\tau * g + (1 - p.\tau) * h \\
&\Leftrightarrow\ ((M \parallel P) \ {}_{p}\!\oplus\ (M \parallel Q)).\tau.f\ .
\end{aligned}$$

Proof of law (L32)

Let P, Q be probabilistic state relations, x a variable, e an expression of suitable type, $p : \Gamma \to (0,1)$ probabilities, τ a state, and f a probabilistic state.

$$\begin{aligned}
&(x := e; (P \ {}_{p}\!\oplus\ Q)).\tau.f \\
&\Leftrightarrow\ (P \ {}_{p}\!\oplus\ Q).([x := e.\tau]\,\tau).f \\
&\Leftrightarrow\ \exists\, g : P.([x := e.\tau]\,\tau),\ h : Q.([x := e.\tau]\,\tau) \bullet \\
&\qquad f = p.([x := e.\tau]\,\tau) * g + (1 - p.([x := e.\tau]\,\tau)) * h \\
&\Leftrightarrow\ \exists\, g : (x := e; P).\tau,\ h : (x := e; Q).\tau \bullet \\
&\qquad f = ([x := e]\,p).\tau * g + (1 - ([x := e]\,p).\tau) * h \\
&\Leftrightarrow\ (x := e; P) \ {}_{[x:=e]\,p}\!\oplus\ (x := e; Q)\ .
\end{aligned}$$

Proof of law (L38)

Let P, Q be extended probabilistic state relations, r, s real-valued expressions, $p : \Gamma \to (0,1)$ probabilities, τ a state, $e \in \mathbb{R}_{\geq 0}$ an expected cost, and f a probabilistic state.

$$\begin{aligned}
&(|\ r\ |; P) \ {}_{p}\!\oplus\ (|\ s\ |; Q).\tau.(e, h) \\
&=\ \exists (c, f) : (|\ r\ |; P).\tau,\ (d, g) : (|\ s\ |; Q).\tau \bullet \\
&\qquad e = c \ {}_{p.\tau}\!\oplus\ d \wedge h = f \ {}_{p.\tau}\!\oplus\ g \\
&=\ \exists (c, f) : P.\tau,\ (d, g) : Q.\tau \bullet \\
&\qquad e = (c + r.\tau) \ {}_{p.\tau}\!\oplus\ (d + s.\tau) \wedge h = f \ {}_{p.\tau}\!\oplus\ g \\
&=\ \exists (c, f) : P.\tau,\ (d, g) : Q.\tau \bullet \\
&\qquad e = (c \ {}_{p.\tau}\!\oplus\ d) + (r.\tau \ {}_{p.\tau}\!\oplus\ s.\tau) \wedge h = f \ {}_{p.\tau}\!\oplus\ g \\
&=\ \exists\, e' \bullet e = (r.\tau \ {}_{p.\tau}\!\oplus\ s.\tau) + e' \wedge \\
&\qquad \exists (c, f) : P.\tau,\ (d, g) : Q.\tau \bullet e' = (c \ {}_{p.\tau}\!\oplus\ d) \wedge h = f \ {}_{p.\tau}\!\oplus\ g
\end{aligned}$$

$$= \ \exists\, e' \bullet e = (r.\tau \ {}_{p.\tau}\!\oplus\ s.\tau) + e' \wedge (P \ {}_p\!\oplus\ Q).\tau.(e', h)$$
$$= \ (|\ r \ {}_p\!\oplus\ s\ |; (P \ {}_p\!\oplus\ Q)).\tau.(e, h) \ .$$

Proof of proposition 4.3

Let $\mathbf{A} = (\Gamma, I, P)$ be a probabilistic action system. We prove first by induction on $n \in \mathbb{N}$:

$$f \in I; (\Downarrow P)^n \ \Leftrightarrow \ (\exists\, t : \mathrm{seq}[n](\mathbb{R}_{\geq 0}) \bullet \mathsf{path}.\mathbf{A}.t.f) \ . \qquad \text{(A.1)}$$

$n = 0$: $f \in I \Leftrightarrow \mathsf{path}.\mathbf{A}.\langle\,\rangle.f$.
$n > 0$:

$$\begin{aligned}
& f \in I; (\Downarrow P)^n \\
\Leftrightarrow\ & f \in I; (\Downarrow P)^{n-1}; (\Downarrow P) \\
\Leftrightarrow\ & \exists\, g,\ M \bullet g \in I; (\Downarrow P)^{n-1} \wedge M \in \mathsf{fun}.(\Downarrow P) \wedge f = g * M \\
\Leftrightarrow\ & \exists\, t : \mathrm{seq}[n-1](\mathbb{R}_{\geq 0}),\ g,\ M \bullet \\
& \quad \mathsf{path}.\mathbf{A}.t.g \wedge M \in \mathsf{fun}.(\Downarrow P) \wedge f = g * M \\
\Leftrightarrow\ & \exists\, t : \mathrm{seq}[n-1](\mathbb{R}_{\geq 0}),\ g,\ C,\ M \bullet \\
& \quad \mathsf{path}.\mathbf{A}.t.g \wedge (C, M) \in \mathsf{fun}.P \wedge f = g * M \\
\Leftrightarrow\ & \exists\, t : \mathrm{seq}[n-1](\mathbb{R}_{\geq 0}),\ g,\ c \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (c, f) \in g * P \\
\Leftrightarrow\ & \exists\, t : \mathrm{seq}[n-1](\mathbb{R}_{\geq 0}),\ c \bullet \mathsf{path}.\mathbf{A}.t^\frown\langle c\rangle.f \\
\Leftrightarrow\ & \exists\, t : \mathrm{seq}[n](\mathbb{R}_{\geq 0}) \bullet \mathsf{path}.\mathbf{A}.t.f \ .
\end{aligned}$$

Next we prove the claim $\mathsf{live}.\mathbf{A} \Leftrightarrow \mathsf{reach}.\mathbf{A} \subseteq \mathsf{dom}.P$ of the proposition:

$$\begin{aligned}
& \mathsf{live}.\mathbf{A} \\
\Leftrightarrow\ & \mathsf{im}.\mathbf{A} = \varnothing \\
\Leftrightarrow\ & \neg\ \exists\, t \bullet \mathsf{im}.\mathbf{A}.t \\
\Leftrightarrow\ & \neg\ \exists\, t, f \bullet \mathsf{path}.\mathbf{A}.t.f \wedge f * P = \varnothing \\
\Leftrightarrow\ & \neg\ \exists\, n, f \bullet f \in I; (\Downarrow P)^n \wedge f * P = \varnothing \\
\Leftrightarrow\ & \forall\, f \bullet (\exists\, n \bullet f \in I; (\Downarrow P)^n) \Rightarrow \mathsf{car}.f \subseteq \mathsf{dom}.P \\
\Leftrightarrow\ & \mathsf{reach}.\mathbf{A} \subseteq \mathsf{dom}.P \ .
\end{aligned}$$

Proof of theorem 4.4

Let $\mathbf{A} = (\Gamma_A, I, P)$ and $\mathbf{C} = (\Gamma_C, J, Q)$ be probabilistic action systems, and let $M \in \mathcal{D}(\Gamma_A, \Gamma_C)$ be a probabilistic state function satisfying (PS1) to (PS3). Lemma A.1 establishes the simulation relationship for traces:

Lemma A.1 For all $t \in \mathrm{seq}(\mathbb{R}_{\geq 0})$ and $f \in \mathbb{D}\,\Gamma_C$:

$$\mathsf{path}.\mathbf{C}.t.f \ \Rightarrow \ (\exists\, g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge f = g * M) \ .$$

To prove theorem 4.4 we have to show $\mathbf{A} \sqsubseteq \mathbf{B}$, i.e. the inclusion of the traces and the impasses.

Trace inclusion. Let t be a trace.

$$
\begin{aligned}
&\mathsf{tr}.\mathbf{C}.t \\
&\quad\Leftrightarrow\ \exists f : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.t.f \\
&\quad\Rightarrow\ \exists f : \mathbb{D}\,\Gamma_C,\ g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge f = g * M \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (\exists f : \mathbb{D}\,\Gamma_C \bullet f = g * M) \\
&\quad\Rightarrow\ \mathsf{tr}.\mathbf{A}.t\ .
\end{aligned}
$$

Impasse inclusion. Let t be a trace.

$$
\begin{aligned}
&\mathsf{im}.\mathbf{C}.t \\
&\quad\Leftrightarrow\ \exists f : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.t.f \wedge f * Q = \varnothing \\
&\quad\Rightarrow\ \exists f : \mathbb{D}\,\Gamma_C,\ g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge f = g * M \wedge f * Q = \varnothing \\
&\quad\Rightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (g * M) * Q = \varnothing \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge g * (M; Q) = \varnothing \\
&\quad\Rightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge g * P = \varnothing && (\mathit{PS2}) \\
&\quad\Leftrightarrow\ \mathsf{im}.\mathbf{A}.t\ .
\end{aligned}
$$

Proof of lemma A.1

Let $t \in \mathrm{seq}(\mathbb{R}_{\geq 0})$ and $f \in \mathbb{D}\,\Gamma_C$. The proof is by induction on the length of trace t.

$$
\begin{aligned}
&\mathsf{path}.\mathbf{C}.\langle\,\rangle.f \\
&\quad\Leftrightarrow f \in J \\
&\quad\Rightarrow f \in I; M && (\mathit{PS1}) \\
&\quad\Leftrightarrow \exists g : \mathbb{D}\,\Gamma_A \bullet g \in I \wedge f = g * M \\
&\quad\Leftrightarrow \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.\langle\,\rangle.g \wedge f = g * M\ .
\end{aligned}
$$

Let $t = s^\frown\langle c\rangle$:

$$
\begin{aligned}
&\mathsf{path}.\mathbf{C}.t.f \\
&\quad\Leftrightarrow\ \exists h : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.s.h \wedge (c, f) \in h * Q \\
&\quad\Rightarrow\ \exists h : \mathbb{D}\,\Gamma_C,\ g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge h = g * M \wedge (c, f) \in h * Q \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c, f) \in (g * M) * Q \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c, f) \in g * (M; Q) \\
&\quad\Rightarrow\ \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c, f) \in g * (P; M) && (\mathit{PS3}) \\
&\quad\Leftrightarrow\ \exists g : \mathbb{D}\,\Gamma_A,\ h : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c, h) \in g * P \wedge f = h * M \\
&\quad\Leftrightarrow\ \exists h : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.h \wedge f = h * M\ .
\end{aligned}
$$

Proof of theorem 4.7

Let $\mathbf{A} = (\Gamma_A, I, P)$ and $\mathbf{C} = (\Gamma_C, J, Q)$ be probabilistic action systems, and $M \in \mathcal{M}(\Gamma_A, \Gamma_C)$ a probabilistic state function satisfying (EQ1) and (EQ2). We need the following lemma:

Lemma A.2 For all $t \in \text{seq}(\mathbb{R}_{\geq 0})$ and $f \in \mathbb{D}\,\Gamma_C$:

$$\mathsf{path}.\mathbf{C}.t.f \;\Leftrightarrow\; (\exists\, g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge f = g * M)\;.$$

Using lemma A.2 we prove trace and impasse identity of the two systems.

Trace identity. Let t be a trace.

$$\begin{aligned}
&\mathsf{tr}.\mathbf{C}.t\\
&\Leftrightarrow\; \exists f : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.t.f\\
&\Leftrightarrow\; \exists f : \mathbb{D}\,\Gamma_C,\; g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge f = g * M\\
&\Leftrightarrow\; \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (\exists f : \mathbb{D}\,\Gamma_C \bullet f = g * M)\\
&\Leftrightarrow\; \mathsf{tr}.\mathbf{A}.t\;,
\end{aligned}$$

where we have used that $\mathsf{dom}.M = \Gamma_A$ in the last step.

Impasse identity. Let t be a trace. Again we rely on M being a total function:

$$\begin{aligned}
&\mathsf{im}.\mathbf{C}.t\\
&\Leftrightarrow\; \exists f : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.t.f \wedge f * Q = \varnothing\\
&\Leftrightarrow\; \exists f : \mathbb{D}\,\Gamma_C,\; g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge f = g * M \wedge f * Q = \varnothing\\
&\Leftrightarrow\; \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge (g * M) * Q = \varnothing\\
&\Leftrightarrow\; \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge g * (M; Q) = \varnothing\\
&\Leftrightarrow\; \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge g * (P; M) = \varnothing && (EQ2)\\
&\Leftrightarrow\; \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.g \wedge g * P = \varnothing\\
&\Leftrightarrow\; \mathsf{im}.\mathbf{A}.t\;.
\end{aligned}$$

Proof of lemma A.2

Let $t \in \text{seq}(\mathbb{R}_{\geq 0})$ and $f \in \mathbb{D}\,\Gamma_C$. The proof is by induction on the length of trace t:

$$\begin{aligned}
&\mathsf{path}.\mathbf{C}.\langle\,\rangle.f\\
&\Leftrightarrow f \in J\\
&\Leftrightarrow f \in I; M && (EQ1)\\
&\Leftrightarrow \exists g : \mathbb{D}\,\Gamma_A \bullet g \in I \wedge f = g * M\\
&\Leftrightarrow \exists g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.\langle\,\rangle.g \wedge f = g * M\;.
\end{aligned}$$

Let $t = s^\frown\langle c\rangle$:

$$
\begin{aligned}
&\mathsf{path}.\mathbf{C}.t.f\\
\Leftrightarrow\;& \exists\, h : \mathbb{D}\,\Gamma_C \bullet \mathsf{path}.\mathbf{C}.s.h \wedge (c,f) \in h * Q\\
\Leftrightarrow\;& \exists\, h : \mathbb{D}\,\Gamma_C,\; g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge h = g * M \wedge (c,f) \in h * Q\\
\Leftrightarrow\;& \exists\, g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,f) \in (g * M) * Q\\
\Leftrightarrow\;& \exists\, g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,f) \in g * (M; Q)\\
\Leftrightarrow\;& \exists\, g : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,f) \in g * (P; M) \qquad (EQ2)\\
\Leftrightarrow\;& \exists\, g : \mathbb{D}\,\Gamma_A,\; h : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.s.g \wedge (c,h) \in g * P \wedge f = h * M\\
\Leftrightarrow\;& \exists\, h : \mathbb{D}\,\Gamma_A \bullet \mathsf{path}.\mathbf{A}.t.h \wedge f = h * M\ .
\end{aligned}
$$

Proof of theorem 4.9

Let $\mathbf{A} = (\Gamma_A, I, P)$ be a probabilistic action system, and let $\phi \in \mathcal{F}(\Gamma)$ be idempotent satisfying (FUS). Let $\mathbf{C} = (\Gamma_A, I;\phi, P;\phi)$ and use theorem 4.7 with $M = \Uparrow\phi$:

$$
\begin{aligned}
I;\phi &= I;\phi\ ,\\
\phi;P;\phi &= P;\phi\ .
\end{aligned}
$$

Proof of proposition 4.10

Let $\mathbf{A} = (\Gamma, I, P)$ be a probabilistic action system, and let $\phi_1, \phi_2 \in \mathcal{F}(\Gamma)$ be fusions of $\mathbf{A}$. Assume $\phi_1;\phi_2 = \phi_2;\phi_1$ holds, then $\phi_1;\phi_2$ is idempotent:

$$
\begin{aligned}
&(\phi_1;\phi_2);(\phi_1;\phi_2)\\
&\quad = \phi_1;(\phi_2;\phi_1);\phi_2\\
&\quad = \phi_1;(\phi_1;\phi_2);\phi_2\\
&\quad = (\phi_1;\phi_1);(\phi_2;\phi_2)\\
&\quad = \phi_1;\phi_2\ .
\end{aligned}
$$

And it satisfies (FUS):

$$
\begin{aligned}
&P;(\phi_1;\phi_2)\\
&\quad = (P;\phi_1);\phi_2\\
&\quad = (\phi_1;P;\phi_1);\phi_2\\
&\quad = (\phi_1;P);(\phi_2;\phi_1)\\
&\quad = \phi_1;(\phi_2;P;\phi_2);\phi_1\\
&\quad = (\phi_1;\phi_2);P;(\phi_1;\phi_2)\ .
\end{aligned}
$$

Proof of proposition 4.11

Let $P, \phi \in \mathcal{E}(\Gamma)$, $P = P_1; P_2$. Assume $P_1; \phi = \phi; P_1; \phi$ and $P_2; \phi = \phi; P_2; \phi$, then:

$$
\begin{aligned}
& P; \phi \\
& \quad = \ (P_1; P_2); \phi \\
& \quad = \ P_1; (\phi; P_2; \phi) \\
& \quad = \ (\phi; P_1; \phi;) P_2; \phi \\
& \quad = \ \phi; P_1; (P_2; \phi) \\
& \quad = \ \phi; P; \phi \ .
\end{aligned}
$$

Proof of proposition 5.1

Let $\mathbf{A} = (\Gamma, I, P)$ be a live probabilistic action system, $t : \mathsf{seq}\, \mathbb{R}_{\geq 0}$ a trace, and $g : \mathbb{D}\, \Gamma$ a probabilistic state. The proof is by induction on the length of trace t. For $t = \langle\, \rangle$:

$$
\begin{aligned}
& \mathsf{path}.\mathbf{A}.\langle\, \rangle.g \\
& \quad \Leftrightarrow \ g \in I \\
& \quad \Leftrightarrow \exists\, f : I \bullet g = f * (\Pi.\langle\, \rangle) \ .
\end{aligned}
$$

For $t = s^\frown \langle c \rangle$:

$$
\begin{aligned}
& \mathsf{path}.\mathbf{A}.t.g \\
& \quad \Leftrightarrow \ \exists\, f \bullet \mathsf{path}.\mathbf{A}.t.f \wedge (c, g) \in f * P \\
& \quad \Leftrightarrow \ \exists\, f \bullet (c, g) \in f * P \wedge \\
& \qquad\qquad \exists\, h : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.t) \bullet \\
& \qquad\qquad f = h * \Pi.\Phi \wedge \\
& \qquad\qquad \forall\, j : \mathsf{dom}.t \bullet t.j = h * \Pi.(\Phi \uparrow j - 1) * \kappa.j \\
& \quad \Leftrightarrow \ \exists\, f \bullet (\exists (C, M) : \mathsf{fun}.P \bullet c = f * C \wedge g = f * M) \wedge \\
& \qquad\quad \exists\, h : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.t) \bullet \\
& \qquad\qquad f = h * \Pi.\Phi \wedge \\
& \qquad\qquad \forall\, j : \mathsf{dom}.t \bullet t.j = h * \Pi.(\Phi \uparrow j - 1) * \kappa.j \\
& \quad \Leftrightarrow \ \exists (C, M) : \mathsf{fun}.P \bullet \\
& \qquad\quad \exists\, f \bullet \exists\, h : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.t) \bullet \\
& \qquad\qquad g = f * M \wedge f = h * \Pi.\Phi \wedge \\
& \qquad\qquad c = f * C \wedge (\forall\, j : \mathsf{dom}.t \bullet t.j = h * \Pi.(\Phi \uparrow j - 1) * \kappa.j) \\
& \quad \Leftrightarrow \ \exists (C, M) : \mathsf{fun}.P \bullet \\
& \qquad\quad \exists\, h : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.t) \bullet \\
& \qquad\qquad g = (h * \Pi.\Phi) * M \wedge
\end{aligned}
$$

$$
\begin{aligned}
& \quad c = (h * \Pi.\Phi) * C \wedge \\
& \quad (\forall j : \mathsf{dom}.t \bullet t.j = h * \Pi.(\Phi \uparrow j-1) * \kappa.j) \\
\Leftrightarrow\ & \exists (C, M) : \mathsf{fun}.P \bullet \\
& \quad \exists\, h : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.t) \bullet \\
& \quad\quad g = h * (\Pi.(\Phi ^\frown \langle M \rangle)) \wedge \\
& \quad\quad (\forall j : \mathsf{dom}.s \bullet s.j = h * \Pi.(\Phi \uparrow j-1) * (\kappa ^\frown \langle C \rangle).j) \\
\Leftrightarrow\ & \exists\, h : I,\ (\kappa, \Phi) : \mathsf{po}.(\mathsf{proc}.\mathbf{A}).(\mathsf{size}.s) \bullet \\
& \quad g = h * (\Pi.\Phi) \wedge \\
& \quad (\forall j : \mathsf{dom}.s \bullet s.j = h * \Pi.(\Phi \uparrow j-1) * \kappa.j)\ .
\end{aligned}
$$

Proof of theorem 5.3

Let $\mathbf{A} = (\Gamma, I, P)$ be live, and $v \in \mathsf{seq}_\infty \mathbb{R}_{\geq 0}$ an infinite trace. First we show

$$
\mathsf{itr}.\mathbf{A}.v \ \Rightarrow\ \exists f : I,\ (\kappa, \Phi) : \mathsf{ipo}.(\mathsf{proc}.\mathbf{A}) \bullet \mathsf{itrace}.f.(\kappa, \Phi).v\ . \tag{A.2}
$$

We remark that there exists an infinite policy of process $\mathsf{proc}.\mathbf{A}$ because $\mathbf{A}$ is live.

Let $v \in \mathsf{itr}.\mathbf{A}$ and t be a prefix of v. Then, by corollary 5.2, there is an initial probabilistic state f_t and a finite policy (κ_t, Φ_t), such that

$$
\mathsf{trace}.f_t.(\kappa_t, \Phi_t).t
$$

Let $f \in \{g \mid (\exists^\infty t \leq v \bullet f_t = g)\}$. We write $\exists^\infty$ for 'there are infinitely many'. Now there is an infinite policy (κ, Φ) such that

$$
\begin{aligned}
(\kappa.k, \Phi.k) \ \in\ \{(C, M) \mid (\exists^\infty t \leq v \bullet (C, M) = (\kappa_t.k, \Phi_t.k) \wedge \\
\mathsf{trace}.f.((\kappa \uparrow k-1)^\frown \langle C \rangle, (\Phi \uparrow k-1)^\frown \langle M \rangle).t\})\ .
\end{aligned}
$$

This policy exists because there are only finitely many (C, M) in $\mathsf{fun}.P$ but infinitely many finite policies (κ_t, Φ_t), i.e. one for each trace t. Once $(\kappa.k, \Phi.k)$ is set, we know that the entire sequence till k occurs infinitely often, so it has infinitely many successors, again chosen from a finite set. At least one of them must occur infinitely often. Then (κ, Φ) satisfies

$$
\forall j : \mathbb{N}_1 \bullet v.j = f * \Pi.(\Phi \uparrow j-1) * \kappa.j\ ,
$$

i.e. claim (A.2) is proven.

The proof of the other implication remains. Let $f \in I$ be an initial probabilistic state, $v \in \mathsf{seq}_\infty \mathbb{R}_{\geq 0}$, and $(\kappa, \Phi) : \mathsf{ipo}.(\mathsf{proc}.\mathbf{A})$ an infinite policy of Markov decision process $\mathsf{proc}.\mathbf{A}$. Assume $\mathsf{itrace}.f.(\kappa, \Phi).v$ holds. This immediately implies

$$
\mathsf{trace}.f.(\kappa \uparrow (\mathsf{size}.t), \Phi \uparrow (\mathsf{size}.t)).t
$$

for all $t \leq v$. From corollary 5.2 it follows $\mathsf{tr}.\mathbf{A}.t$ for all $t \leq v$, in other words, $\mathsf{itr}.\mathbf{A}.v$.

Proof of theorem 5.4

Let $\mathbf{A} = (\Gamma, I, P)$ be live and $v \in \mathsf{itr}.\mathbf{A}$. By theorem 5.3 there is an infinite policy (κ, Φ) (and vice versa) such that

$$\begin{aligned}
&\limsup_{n\to\infty} \frac{1}{n} \sum_{m=1}^{n} v.m \\
&= \limsup_{n\to\infty} \frac{1}{n} \sum_{m=1}^{n} f * \Pi.(\Phi \uparrow m-1) * \kappa.m \\
&= \limsup_{n\to\infty} \frac{1}{n} * f * \sum_{m=1}^{n} \Pi.(\Phi \uparrow m-1) * \kappa.m \\
&= \limsup_{n\to\infty} f * \frac{1}{n} \sum_{m=1}^{n} \Pi.(\Phi \uparrow m-1) * \kappa.m \\
&= f * \limsup_{n\to\infty} \frac{1}{n} \sum_{m=1}^{n} \Pi.(\Phi \uparrow m-1) * \kappa.m \ ,
\end{aligned}$$

where the last equation uses that f has finite carrier. Taking the infimum on both sides we get:

$$\begin{aligned}
&\inf_{v\in\mathsf{itr}.\mathbf{A}} \limsup_{n\to\infty} \frac{1}{n} \sum_{m=1}^{n} v.m \\
&= \inf_{\substack{f\in I,\\ (\kappa,\Phi)\in\mathsf{ipo}.(\mathsf{proc}.\mathbf{A})}} f * \limsup_{n\to\infty} \frac{1}{n} \sum_{m=1}^{n} \Pi.(\Phi \uparrow m-1) * \kappa.m \\
&= \inf_{f\in I} \inf_{(\kappa,\Phi)\in\mathsf{ipo}.(\mathsf{proc}.\mathbf{A})} f * \mathsf{avg}.(\kappa, \Phi) \\
&= \inf_{f\in I} f * \inf_{(\kappa,\Phi)\in\mathsf{ipo}.(\mathsf{proc}.\mathbf{A})} \mathsf{avg}.(\kappa, \Phi) \\
&= \inf_{f\in I} f * \mathsf{opt}.(\mathsf{proc}.\mathbf{A}) \\
&= \min_{f\in I} f * \mathsf{opt}.(\mathsf{proc}.\mathbf{A}) \ .
\end{aligned}$$

Proof of example 5.7

We prove (EQ1) and (EQ2) of theorem 4.7 to establish the equivalence. The program

$$sim \ = \ x := \mathsf{true} \ {}_{\frac{1}{2}}\!\oplus \ x := \mathsf{false}$$

is a suitable probabilistic simulation.

(EQ1) We show the equivalent claim $sim; sim = sim$:

$$\begin{aligned}
∼ sim \\
&= (sim; x := \mathsf{true}) \ {}_{\frac{1}{2}}\!\oplus \ (sim; x := \mathsf{false}) && (L16) \\
&= sim \ . && (L17,L26,L14)
\end{aligned}$$

(EQ2) We have to prove that $sim; A.cycle = B.cycle; sim$ holds for the actions of the two systems:

$$
\begin{aligned}
& sim; A.cycle \\
&= x := \mathsf{true}; A.cycle \;{}_{\frac{1}{2}}\!\oplus\; x := \mathsf{false}; A.cycle && (L17) \\
&= \lfloor 2.0 \rfloor; x := \mathsf{false} \;{}_{\frac{1}{2}}\!\oplus\; x := \mathsf{true} && (C1,C2) \\
&= \lfloor 2.0 \rfloor; x := \mathsf{false} \;{}_{\frac{1}{2}}\!\oplus\; \lfloor 0.0 \rfloor; x := \mathsf{true} && (L36) \\
&= \lfloor \tfrac{1}{2} * 2.0 + \tfrac{1}{2} * 0.0 \rfloor; (x := \mathsf{false} \;{}_{\frac{1}{2}}\!\oplus\; x := \mathsf{true}) && (L38) \\
&= B.cycle; sim
\end{aligned}
$$

$$
\begin{aligned}
& x := \mathsf{true}; A.cycle && \text{(C1)} \\
&= (x := \mathsf{true}; \lfloor x \rfloor; \lfloor 2.0 \rfloor \sqcup x := \mathsf{true}; \lfloor \neg x \rfloor); x := \neg x && (L9) \\
&= (\lfloor \mathsf{true} \rfloor; x := \mathsf{true}; \lfloor 2.0 \rfloor \sqcup \lfloor \mathsf{false} \rfloor; x := \mathsf{true}); x := \neg x && (L28) \\
&= (\lfloor \mathsf{true} \rfloor; \lfloor 2.0 \rfloor \sqcup \lfloor \mathsf{false} \rfloor); x := \mathsf{true}; x := \neg x && (L29,L11) \\
&= (\lfloor 2.0 \rfloor \sqcup \lfloor \mathsf{false} \rfloor); x := \mathsf{false} && (L1,L26) \\
&= \lfloor 2.0 \rfloor; x := \mathsf{false} && (L8)
\end{aligned}
$$

$$
\begin{aligned}
& x := \mathsf{false}; A.cycle && \text{(C2)} \\
&= (x := \mathsf{false}; \lfloor x \rfloor; \lfloor 2.0 \rfloor \sqcup x := \mathsf{false}; \lfloor \neg x \rfloor); x := \neg x && (L9) \\
&= (\lfloor \mathsf{false} \rfloor; x := \mathsf{false}; \lfloor 2.0 \rfloor \sqcup \lfloor \mathsf{true} \rfloor; x := \mathsf{false}); x := \neg x && (L28) \\
&= (\lfloor \mathsf{false} \rfloor; x := \mathsf{false} \sqcup \lfloor \mathsf{true} \rfloor; x := \mathsf{false}); x := \neg x && (L29,L2) \\
&= (\lfloor \mathsf{false} \rfloor \sqcup \lfloor \mathsf{true} \rfloor); x := \mathsf{false}; x := \neg x && (L11) \\
&= \lfloor \mathsf{true} \rfloor; x := \mathsf{true} && (L8,L26) \\
&= x := \mathsf{true} && (L34)
\end{aligned}
$$

Proof of lemma 6.4

Let Γ be state space, $p \in (0,1)$ a probability, $c \in \mathbb{R}_{\geq 0}$ a cost, $\tau \in \Gamma$ a state, and $P \in \mathcal{E}(\Gamma)$ a program. We have to show that:

$$\mathsf{squash}\,(\mathsf{exec}\,\tau\,\tau\,p\,c\,P\,\langle\,\rangle) \;=\; p * (\lfloor c \rfloor; P).\tau\ . \tag{A.3}$$

Let $\varepsilon(P)$ be a *context* of program P consisting of parallel and sequential compositions, such that

$$\mathsf{exec}\,\tau\,\tau'\,p\,c\,\varepsilon(P)\,\langle\,\rangle \;=\; \mathsf{exec}\,\tau\,\tau'\,p\,c\,P\,\rho$$

for some radical stack ρ induced by ε. Note that, if P contains no choice, then neither does $\varepsilon(P)$. Furthermore, the structure of program $\varepsilon(P)$ is such

that P is executed first and afterwards the contents of ε. We prove that for a program P:

$$\begin{aligned} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho) \\ & \quad = \;\; p * (\mid c \mid ; \varepsilon(\psi; (\phi \parallel P))).\tau\;. \end{aligned} \tag{A.4}$$

The two functions ϕ and ψ depend on τ, meaning, they are constant functions that are chosen appropriately. We use substitution notation $[\phi]\,\tau$ with ϕ to denote the corresponding modification $\phi.\tau$ of state τ. Function ϕ is used to "merge" execution branches of parallel compositions. The other function ψ represents an assignment that precedes P (and ϕ). Note that, by definition, the stack $[\![\phi]\!]\,\rho$ might still refer to the initial state τ. The proof of (A.4) is by induction on the structure of $\varepsilon(P)$. Let program *basic* be one of skip, $\mid q \mid$, $\mid r \mid$, $z := e$. We first prove:

$$\begin{aligned} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;\mathit{basic}\,\langle\,\rangle) \\ & \quad = \;\; p * (\mid c \mid ; (\psi; (\phi \parallel \mathit{basic}))).\tau\;. \end{aligned} \tag{A.5}$$

There are four cases to prove. We begin with program skip:

$$\begin{aligned} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;\mathsf{skip}\,\langle\,\rangle) \\ & \quad = \;\; \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;\Lambda\,\langle\,\rangle) \\ & \quad = \;\; \mathsf{squash}\,(([\phi]\,[\psi]\,\tau)\;@\;p : c) \\ & \quad = \;\; p * \{c, ([\phi]\,[\psi]\,\tau)\;@\;1\} \\ & \quad = \;\; p * (\mid c \mid ; (\psi; (\phi \parallel \mathsf{skip}))).\tau \end{aligned}$$

The proof for the guard statement is similar:

$$\begin{aligned} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c \mid q \mid \langle\,\rangle) \\ & \quad = \;\; \mathsf{squash}\,(\textbf{if}\; q.([\psi]\,\tau)\;\textbf{then}\;\mathsf{exec}\,\tau\,([\phi]\,[\psi]\,\tau)\,p\;c\;\Lambda\,\langle\,\rangle\;\textbf{else}\;\mathsf{null}) \\ & \quad = \;\; \begin{cases} p * \{c, ([\phi]\,[\psi]\,\tau)\;@\;1\} & \text{if } q.([\psi]\,\tau) \\ p * \varnothing & \text{if } \neg\, q.([\psi]\,\tau) \end{cases} \\ & \quad = \;\; p * (\mid c \mid ; \mid q.([\psi]\,\tau) \mid ; (\psi; \phi)).\tau \\ & \quad = \;\; p * (\mid c \mid ; (\psi; (\phi \parallel \mid q.\tau \mid))).\tau \\ & \quad = \;\; p * (\mid c \mid ; (\psi; (\phi \parallel \mid q \mid))).\tau \end{aligned}$$

In the case of the cost statement we have:

$$\begin{aligned} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c \mid r \mid \langle\,\rangle) \\ & \quad = \;\; \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,(c + r.([\psi]\,\tau))\,\Lambda\,\langle\,\rangle) \\ & \quad = \;\; \mathsf{squash}\,(([\phi]\,[\psi]\,\tau)\;@\;p : c + r.([\psi]\,\tau)) \\ & \quad = \;\; p * \{c + r.([\psi]\,\tau), ([\phi]\,[\psi]\,\tau)\;@\;1\} \\ & \quad = \;\; p * (\mid c + r.([\psi]\,\tau) \mid ; (\psi; \phi)).\tau \\ & \quad = \;\; p * (\mid c \mid ; (\psi; (\phi \parallel \mid r.([\psi]\,\tau) \mid))).\tau \\ & \quad = \;\; p * (\mid c \mid ; (\psi; (\phi \parallel \mid r \mid))).\tau \end{aligned}$$

In the next case, note that the two substitutions $[x := e.([\psi]\,\tau)]$ and $[\phi]$ change different variables. In other words we assume that the parallel component ϕ does not alter variable x, i.e. $x.\tau = x.([\phi]\,\tau)$. However, ψ may alter variable x. So the order in which $[x := e.([\psi]\,\tau)]$ and $[\psi]$ are applied is important.

$$\begin{aligned}
&\mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,c\,(x := e)\,\langle\,\rangle)\\
&\quad = \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([x := e.([\psi]\,\tau)]\,[\phi]\,[\psi]\,\tau)\,p\,c\,\Lambda\,\langle\,\rangle)\\
&\quad = \mathsf{squash}\,(([\phi]\,[x := e.([\psi]\,\tau)]\,[\psi]\,\tau)\ @\ p : c)\\
&\quad = p * \{c, ([\phi]\,[x := e.([\psi]\,\tau)]\,[\psi]\,\tau)\ @\ 1\}\\
&\quad = p * (|\ c\ |; \psi; x := e.([\psi]\,\tau); \phi).\tau\\
&\quad = p * (|\ c\ |; (\psi; (\phi \parallel x := e))).\tau
\end{aligned}$$

This proves claim (A.5).

In the next step we prove (A.5) for an enlarged program $\varepsilon'(Q)$ where either $\varepsilon'(Q) = (Q; P)$ or $\varepsilon'(Q) = (Q \parallel P)$, and program P being a choice-free program:

$$\begin{aligned}
&\mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,c\,(\varepsilon'(\mathit{basic}))\,([\![\phi]\!]\,[\![\psi]\!]\,\rho))\\
&\quad = p * (|\ c\ |; \varepsilon(\psi; (\phi \parallel \varepsilon'(\mathit{basic})))).\tau\ .
\end{aligned} \tag{A.6}$$

By (A.5) this holds for the empty context of a *basic* program. Assume it holds for $\varepsilon(P)$. We do not consider skip anymore because it is subsumed by the guard construct:

$$\begin{aligned}
&\mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,c\,(|\ q\ |; P)\,([\![\phi]\!]\,[\![\psi]\!]\,\rho))\\
&\quad = \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,c\,|\ q\ |\,((((([\psi]\,\tau); P) \to ([\![\phi]\!]\,[\![\psi]\!]\,\rho)))\\
&\quad = \mathsf{squash}\,(\mathbf{if}\ q.([\psi]\,\tau)\\
&\qquad\quad \mathbf{then}\ \mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,c\,\Lambda\,(((([\psi]\,\tau); P) \to ([\![\phi]\!]\,[\![\psi]\!]\,\rho))\\
&\qquad\quad \mathbf{else}\ \mathsf{null})\\
&\quad = \mathsf{squash}\,(\mathbf{if}\ q.([\psi]\,\tau)\\
&\qquad\quad \mathbf{then}\ \mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,c\,P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)\\
&\qquad\quad \mathbf{else}\ \mathsf{null})\\
&\quad = \begin{cases} p * (|\ c\ |; \varepsilon(\psi; (\phi \parallel P))).\tau & \text{if } q.([\psi]\,\tau)\\ p * \varnothing & \text{if } \neg\, q.([\psi]\,\tau)\end{cases}\\
&\quad = p * (|\ [\psi]\,q\ |; |\ c\ |; \varepsilon(\psi; (\phi \parallel P))).\tau\\
&\quad = p * (|\ c\ |; \varepsilon(\psi; (\phi \parallel (|\ q\ |; P)))).\tau
\end{aligned}$$

In the last equation we used the fact that $\varepsilon(P)$ is choice-free. Similarly to the guard statement we prove for the cost statement:

$$\mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,c\,(|\ r\ |; P)\,([\![\phi]\!]\,[\![\psi]\!]\,\rho))$$

$$
\begin{aligned}
&= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c \mid r \mid ((([\psi]\,\tau); P) \rightarrow (\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho)))\\
&= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,(c + r.([\psi]\,\tau))\,\Lambda\\
&\qquad\qquad ((([\psi]\,\tau); P) \rightarrow (\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho)))\\
&= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,(c + r.([\psi]\,\tau))\,P\,(\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho))\\
&= p * (|\ c + r.([\psi]\,\tau)\ |; \varepsilon(\psi; (\phi \parallel P))).\tau\\
&= p * (|\ c\ |; |\ [\psi]\,r\ |; \varepsilon(\psi; (\phi \parallel P))).\tau\\
&= p * (|\ c\ |; \varepsilon(\psi; (\phi \parallel (|\ r\ |; P)))).\tau
\end{aligned}
$$

In the next case we use that in $\varepsilon(x := e; P)$ the assignment $x := e$ is executed first, and also that $\varepsilon(P)$ is choice-free:

$$
\begin{aligned}
&\mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\,(x := e; P)\,(\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho))\\
&\quad= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\,(x := e)\\
&\qquad\qquad ((([\psi]\,\tau); P) \rightarrow (\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho)))\\
&\quad= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([x := e.([\psi]\,\tau)]\,[\phi]\,[\psi]\,\tau)\,p\;c\,\Lambda\\
&\qquad\qquad (\llbracket x := e.([\psi]\,\tau)\rrbracket\,((([\psi]\,\tau; P) \rightarrow (\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho)))))\\
&\quad= \mathsf{squash}\,(\mathsf{exec}\,([x := e.([\psi]\,\tau)]\,[\psi]\,\tau)\,([\phi]\,[x := e.([\psi]\,\tau)]\,[\psi]\,\tau)\,p\;c\,P\\
&\qquad\qquad (\llbracket\phi\rrbracket\,\llbracket x := e.([\psi]\,\tau)\rrbracket\,\llbracket\psi\rrbracket\,\rho))\\
&\quad= p * (|\ c\ |; \varepsilon(\psi; x := ([\psi]\,e); (\phi \parallel P))).\tau\\
&\quad= p * (|\ c\ |; \varepsilon(\psi; (\phi \parallel (x := e; P)))).\tau
\end{aligned}
$$

Next we deal with a context that is enlarged by parallel composition. We only consider the cost and the assignment statement. The case of the guard statement is proven similarly to the cost statement. We begin with the cost statement:

$$
\begin{aligned}
&\mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\,(|\ r\ | \parallel P)\,(\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho))\\
&\quad= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c \mid r \mid ((([\psi]\,\tau) \parallel P) \rightarrow (\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho)))\\
&\quad= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,(c + r.([\psi]\,\tau))\,\Lambda\\
&\qquad\qquad ((([\psi]\,\tau) \parallel P) \rightarrow (\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho)))\\
&\quad= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\,(c + r.([\psi]\,\tau))\,P\,(\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho))\\
&\quad= p * (|\ c + r.([\psi]\,\tau)\ |; \varepsilon(\psi; (\phi \parallel P))).\tau\\
&\quad= p * (|\ c\ |; |\ [\psi]\,r\ |; \varepsilon(\psi; (\phi \parallel P))).\tau\\
&\quad= p * (|\ c\ |; \varepsilon(\psi; (\phi \parallel |\ r\ | \parallel P))).\tau
\end{aligned}
$$

Next is the assignment statement:

$$
\begin{aligned}
&\mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\,(x := e \parallel P)\,(\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho))\\
&\quad= \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\,(x := e)\\
&\qquad\qquad ((([\psi]\,\tau) \parallel P) \rightarrow (\llbracket\phi\rrbracket\,\llbracket\psi\rrbracket\,\rho)))
\end{aligned}
$$

$$
\begin{aligned}
= {} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([x := e.([\psi]\,\tau)]\,[\phi]\,[\psi]\,\tau)\,p\;c\;\Lambda \\
& \qquad (((([\psi]\,\tau) \parallel P) \rightarrow ([\![x := e.([\psi]\,\tau)]\!]\,[\![\phi]\!]\,[\![\psi]\!]\,\rho))) \\
= {} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([x := e.([\psi]\,\tau)]\,[\phi]\,[\psi]\,\tau)\,p\;c\;P \\
& \qquad ([\![x := e.([\psi]\,\tau)]\!]\,[\![\phi]\!]\,[\![\psi]\!]\,\rho)) \\
= {} & p * (|\; c \;|; \varepsilon(\psi; (\phi \parallel x := ([\psi]\,e) \parallel P)).\tau \\
= {} & p * (|\; c \;|; \varepsilon(\psi; (\phi \parallel x := e \parallel P)).\tau
\end{aligned}
$$

Assume now that (A.4) holds for a general context ε containing choices, and programs P and Q. Then it holds also for $P \sqcup Q$ and $P \;{}_a\!\oplus\; Q$. We treat nondeterministic choice first:

$$
\begin{aligned}
& \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\,(P \sqcup Q)\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) \\
= {} & \mathsf{squash}\,(\\
& \quad \textbf{if}\ \mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho) = \mathsf{null} \\
& \quad \textbf{and}\ \mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;Q\,([\![\phi]\!]\,[\![\psi]\!]\,\rho) = \mathsf{null} \\
& \qquad \textbf{then}\ \mathsf{null} \\
& \qquad \textbf{else}\ (\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) \\
& \qquad\qquad \Box \\
& \qquad\qquad (\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;Q\,([\![\phi]\!]\,[\![\psi]\!]\,\rho))) \\
= {} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) \cup \\
& \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\;Q\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) \\
= {} & (p * (|\; c \;|; \varepsilon(\psi; (\phi \parallel P)).\tau) \cup (p * (|\; c \;|; \varepsilon(\psi; (\phi \parallel Q)).\tau) \\
= {} & p * ((|\; c \;|; \varepsilon(\psi; (\phi \parallel P)).\tau) \cup (|\; c \;|; \varepsilon(\psi; (\phi \parallel Q)).\tau)) \\
= {} & p * ((|\; c \;|; \varepsilon(\psi; (\phi \parallel P))) \sqcup (|\; c \;|; \varepsilon(\psi; (\phi \parallel Q))).\tau) \\
= {} & p * (|\; c \;|; \varepsilon(\psi; (\phi \parallel (P \sqcup Q))))
\end{aligned}
$$

In the last equation we have used law (L9) with $R = |\; c \;|$, laws (L11) and (L13) for ε, again law (L9) but with $R = \psi$, and law (L13) with $P = \phi$.

Next we regard probabilistic choice:

$$
\begin{aligned}
& \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,p\;c\,(P \;{}_a\!\oplus\; Q)\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) \\
= {} & \mathsf{squash}\,(\\
& \quad \textbf{if}\ \mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,(p * a.([\psi]\,\tau))\;c\;P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho) = \mathsf{null} \\
& \quad \textbf{or}\ \mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,(p * (1 - a.([\psi]\,\tau)))\;c\;Q\,([\![\phi]\!]\,[\![\psi]\!]\,\rho) = \mathsf{null} \\
& \qquad \textbf{then}\ \mathsf{null} \\
& \qquad \textbf{else}\ (\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,(p * a.([\psi]\,\tau))\;c\;P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) \\
& \qquad\qquad \oplus \\
& \qquad\qquad (\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,(p * (1 - a.([\psi]\,\tau)))\;c\;Q\,([\![\phi]\!]\,[\![\psi]\!]\,\rho))) \\
= {} & \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,(p * a.([\psi]\,\tau))\;c\;P\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) +
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{squash}\,(\mathsf{exec}\,([\psi]\,\tau)\,([\phi]\,[\psi]\,\tau)\,(p*(1-a.([\psi]\,\tau)))\,c\,Q\,([\![\phi]\!]\,[\![\psi]\!]\,\rho)) \\
= \; & ((p*a.([\psi]\,\tau))*(|\,c\,|;\varepsilon(\psi;(\phi \parallel P)).\tau)+ \\
& ((p*(1-a.([\psi]\,\tau)))*(|\,c\,|;\varepsilon(\psi;(\phi \parallel Q)).\tau) \\
= \; & p*((|\,c\,|;\varepsilon(\psi;(\phi \parallel P)))\ {}_{a.([\psi]\,\tau)}\!\oplus\ (|\,c\,|;\varepsilon(\psi;(\phi \parallel Q)))).\tau \\
= \; & p*(|\,c\,|;\varepsilon(\psi;(\phi \parallel (P\ {}_{a.([\psi]\,\tau)}\!\oplus\ Q)))).\tau \\
= \; & p*(|\,c\,|;\varepsilon(\psi;(\phi \parallel (P\ {}_{a}\!\oplus\ Q)))).\tau
\end{aligned}
$$

In the last but one equation we have used law (L38), laws (L17) and (L19) with ε, law (L16) with $P = \psi$, and law (L18) with $P = \phi$.

Claim (A.3) follows from (A.4) with $\rho = \langle\,\rangle$, $\phi = \mathsf{id}$, and $\psi = \mathsf{id}$.

Proof of proposition 7.1

Let $p_i \in (0,1)$ for all $i \in I$, and all $M_i \in \mathcal{M}(\Gamma)$ non-interfering. We prove the claim by induction on the size of I.

$$
\begin{aligned}
& |||\, i : \varnothing \bullet (p_i\ ?\ M_i) \\
& \quad = \mathsf{skip} \\
& \quad = \textstyle\bigoplus J : \mathbb{P}\,\varnothing \mid (\prod i : J \bullet p_i) * (\prod i : \varnothing \setminus J \bullet 1 - p_i) \bullet \mathsf{skip} \\
& \quad = \textstyle\bigoplus J : \mathbb{P}\,\varnothing \mid (\prod i : J \bullet p_i) * (\prod i : \varnothing \setminus J \bullet 1 - p_i) \bullet (|||\, j : J \bullet M_j)\ .
\end{aligned}
$$

And for $\mathsf{card}.I > 0$, where $I = K \cup \{k\}$, $k \notin K$:

$$
\begin{aligned}
& |||\, i : I \bullet (p_i\ ?\ M_i) \\
& \quad = (|||\, i : K \bullet M_i); (p_k\ ?\ M_k) \\
& \quad = \textstyle\bigoplus J : \mathbb{P}\,K \mid (\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) \bullet \\
& \qquad\quad (|||\, j : J \bullet M_j); \\
& \qquad (p_k\ ?\ M_k) \\
& \quad = \textstyle\bigoplus J : \mathbb{P}\,K \mid (\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) \bullet \\
& \qquad\quad (|||\, j : J \bullet M_j); \\
& \qquad (M_k\ {}_{p_k}\!\oplus\ \mathsf{skip}) \\
& \quad = \textstyle\bigoplus J : \mathbb{P}\,K \mid (\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) \bullet \\
& \qquad\quad (|||\, j : J \bullet M_j); M_k \\
& \qquad {}_{p_k}\!\oplus \\
& \qquad \textstyle\bigoplus J : \mathbb{P}\,K \mid (\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) \bullet \\
& \qquad\quad (|||\, j : J \bullet M_j); \mathsf{skip} \\
& \quad = \textstyle\bigoplus J : \mathbb{P}\,K \mid (\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) \bullet \\
& \qquad\quad (|||\, j : J \cup \{k\} \bullet M_j) \\
& \qquad {}_{p_k}\!\oplus \\
& \qquad \textstyle\bigoplus J : \mathbb{P}\,K \mid (\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) \bullet \\
& \qquad\quad (|||\, j : J \bullet M_j)
\end{aligned}
$$

Let $\tau \in \Gamma$, $f \in \mathbb{D}\,\Gamma$, and for $J \subseteq I$ let $f_J \in \mathbb{D}\,\Gamma$ such that

$$\{f_J\} \ = \ (|||\, j : J \bullet M_j).\tau \ .$$

Thus

$$\begin{aligned}
&|||\, i : I \bullet (p_i \ ?\ M_i).\tau.f \\
&\quad \Leftrightarrow \ f = p_k * \textstyle\sum_{J \subseteq K}(\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) * f_{J \cup \{k\}} + \\
&\qquad\qquad (1 - p_k) * \textstyle\sum_{J \subseteq K}(\prod i : J \bullet p_i) * (\prod i : K \setminus J \bullet 1 - p_i) * f_J \\
&\quad \Leftrightarrow \ f = \textstyle\sum_{J \subseteq K} (\prod i : J \cup \{k\} \bullet p_i) * \\
&\qquad\qquad\qquad (\prod i : I \setminus (J \cup \{k\}) \bullet 1 - p_i) * f_{J \cup \{k\}} + \\
&\qquad\qquad \textstyle\sum_{J \subseteq K}(\prod i : J \bullet p_i) * (\prod i : I \setminus J \bullet 1 - p_i) * f_J \\
&\quad \Leftrightarrow \ \textstyle\sum_{J \subseteq I}(\prod i : J \bullet p_i) * (\prod i : I \setminus J \bullet 1 - p_i) * f_J \\
&\quad \Leftrightarrow \ (\bigoplus J : \mathbb{P}\, I \mid (\prod i : J \bullet p_i) * (\prod i : I \setminus J \bullet 1 - p_i) \bullet (|||\, j : J \bullet M_j)).\tau.f \ .
\end{aligned}$$

Proof of proposition 7.2

Let I be a finite set and all $R_i \in \mathcal{R}(\Gamma)$ be non-interfering. The proof of the claim is by induction on the size of I.

$$\begin{aligned}
&|||\, i : \varnothing \bullet (\mathsf{skip} \sqcup R_i) \\
&\quad = \ \mathsf{skip} \\
&\quad = \ \textstyle\bigsqcup J : \mathbb{P}\,\varnothing \bullet \mathsf{skip} \\
&\quad = \ \textstyle\bigsqcup J : \mathbb{P}\,\varnothing \bullet (|||\, j : J \bullet R_j) \ .
\end{aligned}$$

And for $\mathsf{card}.I > 0$, where $I = K \cup \{k\}$, $k \notin K$:

$$\begin{aligned}
&|||\, i : I \bullet (\mathsf{skip} \sqcup R_i) \\
&\quad = \ (|||\, i : K \bullet (\mathsf{skip} \sqcup R_i)); (\mathsf{skip} \sqcup R_k) \\
&\quad = \ (\textstyle\bigsqcup J : \mathbb{P}\, K \bullet (|||\, j : J \bullet R_j)); (\mathsf{skip} \sqcup R_k) \\
&\quad = \ (\textstyle\bigsqcup J : \mathbb{P}\, K \bullet (|||\, j : J \bullet R_j); \mathsf{skip}) \sqcup \\
&\qquad\ \ (\textstyle\bigsqcup J : \mathbb{P}\, K \bullet (|||\, j : J \bullet R_j); R_k) \\
&\quad = \ (\textstyle\bigsqcup J : \mathbb{P}\, K \bullet (|||\, j : J \bullet R_j)) \sqcup \\
&\qquad\ \ (\textstyle\bigsqcup J : \mathbb{P}\, K \bullet (|||\, j : J \cup \{k\} \bullet R_j)) \\
&\quad = \ \textstyle\bigsqcup J : \mathbb{P}\, K \bullet (|||\, j : J \bullet R_j) \ .
\end{aligned}$$

Proof of proposition 7.3

Let $v \in \mathsf{itr}.\mathbf{A}$ be an infinite trace. For even $n = 2m$:

$$\lim_{m \to \infty} \frac{2}{2m} \sum_{i=1}^{2m} \sigma.v.i \ = \ \lim_{m \to \infty} \frac{1}{m} \sum_{i=1}^{2m} \sigma.v.i \ = \ \lim_{m \to \infty} \frac{1}{m} \sum_{i=1}^{m} v.i \ .$$

Now let $n = 2m + 1$:

$$
\begin{aligned}
&\lim_{m\to\infty} \frac{2}{2m+1} \sum_{i=1}^{2m+1} \sigma.v.i \\
&= \lim_{m\to\infty} \frac{2}{2m+1-(2-1)} \sum_{i=2}^{2m+1} \sigma.v.i \\
&= \lim_{m\to\infty} \frac{2}{2m} \sum_{i=3}^{2m+2} \sigma.v.i \\
&= \lim_{m\to\infty} \frac{1}{m} \sum_{i=3}^{2(m+1)} \sigma.v.i \\
&= \lim_{m\to\infty} \frac{1}{m} \sum_{i=2}^{m+1} v.i \\
&= \lim_{m\to\infty} \frac{1}{(m+1)-(2-1)} \sum_{i=2}^{m+1} v.i \\
&= \lim_{m\to\infty} \frac{1}{m+1} \sum_{i=1}^{m+1} v.i \\
&= \lim_{m\to\infty} \frac{1}{m} \sum_{i=1}^{m} v.i \ ,
\end{aligned}
$$

where we have used proposition 5.8 two times.

Appendix B

Mathematical Notation

Functions and Relations

$r^{\sim}$	$\{(y, x) \mid (x, y) \in r\}$	(relational inverse)
$r[X]$	$\{y \mid (\exists\, x \bullet (x, y) \in r)\}$	(relational image)
$X \lhd r$	$\{(x, y) \mid x \in X \wedge (x, y) \in r\}$	(domain restriction)
$X ⩤ r$	$\{(x, y) \mid x \notin X \wedge (x, y) \in r\}$	(domain subtraction)
$r \rhd Y$	$\{(x, y) \mid y \in Y \wedge (x, y) \in r\}$	(range restriction)
$r ⩥ Y$	$\{(x, y) \mid y \notin Y \wedge (x, y) \in r\}$	(range subtraction)
$r \mathbin{\lhd\!\!\!+} r'$	$(\mathsf{dom}.r' ⩤ r) \cup r'$	(left overriding)
$r' \mathbin{+\!\!\!\rhd} r$	$r \mathbin{\lhd\!\!\!+} r'$	(right overriding)
$\lambda\, x : X \bullet (q \mid e)$	$\{x \mapsto y \mid q.x \wedge e.x = y\}$	(lambda abstraction)
$\lambda\, x : X \bullet e$	$(\lambda\, x : X \bullet (\mathsf{true} \mid e))$	(lambda abstraction)

Sequences

$\mathsf{size}.s$	$\mathsf{card}.(\mathsf{dom}.s)$	(size)
$s \uparrow n$	$(1 \,..\, n) \lhd s$	(retain)
$s \downarrow n$	$\{(m - n) \mapsto x \mid m > n \wedge s.m = x\}$	(remove)
$s ^\frown t$	$s \cup \{(n + \mathsf{size}.s) \mapsto x \mid s.n = x\}$	(concat)
$x \rightarrow s$	$\langle x \rangle ^\frown s$	(prepend)
$s \leftarrow x$	$s ^\frown \langle x \rangle$	(append)
$s \leq t$	$(\exists\, n \bullet s = t \uparrow n)$	(prefix)

Arithmetics

In specifications we mix expressions of type $\mathbb{R}$ and $\mathbb{N}$ when the resulting number is of type $\mathbb{R}$. Mathematically this makes no difference for division-free expressions. In real expressions division is always real division. We have decided to use this convention to make formulas more readable. Otherwise explicit type casts would be needed (as they are present in the ASCII-based textual form used with the software tool of chapter 6). We use sum and product of numbers in the usual mathematical way. However in specification we stick to the B-notation:

$\sum x : X \bullet e$	$\sum_{x:X} e.x$	(sum)
$\sum x : X \bullet (q \mid e)$	$\sum x : \{x' \mid x' \in X \wedge q.x'\} \bullet e.x$	(selective sum)
$\prod x : X \bullet e$	$\prod_{x:X} e.x$	(product)
$\prod x : X \bullet (q \mid e)$	$\prod x : \{x' \mid x' \in X \wedge q.x'\} \bullet e.x$	(selective product)

Appendix C

ASCII-Representation of Probabilistic Action Systems

Sets

`m..n` (interval)
`(x, y)` (pair)
`x |-> y` (pair)
`{x : S | P}` (set comprehension)
`{x : S, y : T | P}` (set comprehension)
$\{x_1, x_2, \ldots, x_n\}$ (set enumeration)
`{}` (empty set)
`pow(S)` (power set)
`pow1(S)` (non-empty subsets)
`S >< T` (product set)
`S \/ T` (set union)
`S /\ T` (set intersection)
`S \ T` (set difference)
`x : S` (set membership)
`x /: S` (set non-membership)
`x = y` (equality)
`x /= y` (inequality)
`S <: T` (set inclusion)
`S /<: T` (set non-inclusion)
`S <<: T` (strict set inclusion)
`S /<<: T` (strict set non-inclusion)
`card(S)` (set cardinality)

Predicates

!x_1 : S_1, x_2 : S_2, . . . , x_n : S_n . P (for all)

#x_1 : S_1, x_2 : S_2, . . . , x_n : S_n . P (there is)

P or Q (disjunction)

P & Q (conjunction)

not(P) (negation)

P => Q (implication)

P <=> Q (equivalence)

Functions and Relations

S --> T (total function type)

S +-> T (partial function type)

S <-> T (relation type)

f(x) (function application)

%x : S . (P | E) (lambda abstraction)

%x : S, y : T . (P | E) (lambda abstraction)

R[S] (relational image)

S <| R (domain restriction)

S <<| R (domain subtraction)

R |> S (range restriction)

R |>> S (range subtraction)

R <+ Q (left overriding)

R +> Q (right overriding)

dom(R) (domain)

ran(R) (range)

R~ (inverse)

id(S) (identity)

Sequences

seq[n](S) (sequences of max. length n)

iseq[n](S) (injective sequences of max. length n)

S <= T (prefix)

S < T (strict prefix)

S <- x (append)

x -> S (prepend)

<$x_1, x_2, \ldots, x_n$> (sequence enumeration)

<> (empty sequence)

S^T (prepend)

x /|\ S	(retain)
x \\|/ S	(remove)
rev(S)	(reverse)
first(S)	(first)
last(S)	(last)
front(S)	(front)
tail(S)	(tail)
size(S)	(size)

Arithmetics

+!x : S.(P | E)	(set summation)
+!x : S, y : T.(P | E)	(set summation)
*!x : S.(P | E)	(set product)
*!x : S, y : T.(P | E)	(set product)
$(x)	(type cast)
min(x)	(minimum)
max(x)	(maximum)
x + y	(addition)
x − y	(subtraction)
x * y	(multiplication)
x / y	(division)

Program Constructs

skip	(skip statement)
[P]	(guard statement)
[E]	(cost statement)
x := E	(assignment)
$x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n$	(simultaneous assignment)
A || B	(parallel composition)
A ; B	(sequential composition)
A [] B	(nondeterministic choice)
@ x : S . A	(finite nondeterministic choice)
A [[E]] B	(probabilistic choice)
?x : S | E . A	(finite probabilistic choice)

Index

aggregation . . . 57
bias (of policy) . . . 94
exec . . . 91–93
fusion . . . 58
gain (of policy) . . . 94
game . . . 91
game tree . . . 89
  well-formed . . . 89
ground . . . 84
hash function . . . 86
idempotent . . . 57
Markov decision process . . . 68
  aperiodic . . . 97
  avg (average cost) . . . 72
  connected . . . 73
  fin (finite-horizon) . . . 71
  ipo (infinite policy) . . . 70
  itrace (infinite trace) . . . 70
  multichain . . . 94
  opt (average cost optimal) 72
  po (finite policy) . . . 69
  proc . . . 69
  rab (discounted cost) . . . 71
  stationary . . . 70
  tot (total expected cost) . . 71
  trace (finite trace) . . . 69
  unichain . . . 94
play . . . 87, 89
probabilistic action system . . . 45
  $\mathbf{A} \equiv \mathbf{C}$ . . . 50
  $\mathbf{A} \sqsubseteq \mathbf{C}$ . . . 50
  action . . . 45, 86
  beh . . . 45
  connected . . . 73
  im (impasse) . . . 46
  initialisation . . . 45, 86
  live . . . 50
  path . . . 45
  reach . . . 50
  tr (trace) . . . 45
  val (optimal value) . . . 72
probabilistic simulation . . . 51
probabilistic state . . . 25
  $\bigoplus_{i:I} p_i \bullet f_i$ . . . 26
  $\mathbb{D}\,\Gamma$ . . . 25
  $f\ {}_p\!\oplus\ g$ . . . 25
  $f + g$ . . . 25
  $p * f$ . . . 25
  car (carrier) . . . 25
  expectation . . . 26
probabilistic state function . . . 27
  $M * C$ . . . 27
  $M; N$ . . . 28
  $\mathcal{D}(\Gamma, \Gamma')$ . . . 28
  $\mathcal{M}(\Gamma, \Gamma')$ . . . 27
  $\Pi.\Phi$ (sequential product) . 28
  $\Uparrow\phi$ . . . 28
  $f * M$ . . . 27
  $p\ ?\ M$ . . . 106
probabilistic state relation . . . 29
  $P \sqcup Q$ . . . 30
  $P\ {}_p\!\oplus\ Q$ . . . 30
  $P; Q$ . . . 31
  P^n (finite iteration) . . . 32
  $P_1 \parallel P_2$ . . . 30
  $\bigsqcup i : I \bullet Q_i$ . . . 30
  $||| \; i : I \bullet P_i$ . . . 106
  $\bigoplus i : I \mid p_i \bullet P_i$ . . . 30
  $\mathcal{P}(\Gamma, \Gamma')$ . . . 29
  $\Uparrow M$ . . . 32
  $\Uparrow R$ . . . 29
  LET $x = E$ **IN** P . . . 109

non-interfering . . . 106
probabilistic state relation
extended . . . 34
$P \sqcup Q$ . . . 34
$P \parallel Q$ . . . 36
$P \;{}_p\!\oplus\; Q$ . . . 34
$P; Q$ . . . 35
$\bigoplus i : I \mid p_i \bullet P_i$ . . . 35
$\Downarrow P$ . . . 34
$\mathcal{E}(\Gamma, \Gamma')$ . . . 34
$\mid e \mid$ (cost) . . . 34
$\Uparrow P$ . . . 34
cost . . . 33
$\bigoplus_{i \in I} p_i \bullet c_i$ . . . 33
$c_1 \;{}_p\!\oplus\; c_2$ . . . 33
fun . . . 35
rad . . . *see* radical
radical . . . 91
parallel . . . 91
sequential . . . 91
stack . . . 91
squash . . . 93
$X + Y$ . . . 93
$p * X$ . . . 93
squash game . . . 93
state . . . 22, 86
state function . . . 22
$\mathcal{F}(\Gamma, \Gamma')$ . . . 22
id . . . 22
skip . . . 22
state relation . . . 23
$R_1 \sqcup R_2$ . . . 23
$R_1 \parallel R_2$ . . . 23
$R_1; R_2$ . . . 23
$\bigsqcup i : I \bullet R_i$ . . . 23
$\mid q \mid$ (guard) . . . 23
$\pi_i := e$ . . . 23
$\mathcal{R}(\Gamma, \Gamma')$ . . . 23
$\Uparrow \phi$ . . . 24
$x_1, x_2 := e_1, e_2$ . . . 24
deterministic . . . 24
stop . . . 23
state space . . . 22
state variable . . . 22
time slot . . . 45
transition . . . 45
period . . . 45
type-checking . . . 81–85
unify . . . 84

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[3] Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in B. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag, 1998.

[4] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Systems.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, New York, 1991.

[5] R.-J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[6] R.-J. R. Back and J. von Wright. Duality in specification languages: a lattice theoretical approach. *Acta Informatica*, 27:583–625, 1990.

[7] Ralph-Johan Back and Michael Butler. Fusion and simultaneous execution in the refinement calculus. *Acta Informatica*, 35:921–949, 1998.

[8] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. Reports on Mathematics and Computer Science 153, Åbo Akademi, 1994.

[9] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[10] M. Bernardo, W.R. Cleaveland, S.T. Sims, and W.J. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In S. Budkowski, A. Cavalli, and E. Najm, editors,

Proc. of the IFIP Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing, and Verification (FORTE/PSTV 1998), pages 457–467, Paris, France, 1998. Kluwer.

[11] M. Bernardo, L. Donatiello, and R. Gorrieri. Operational GSPN semantics of MPA. Technical Report BOLOGNA#UBLCS-94-12, University of Bologna (Italy). Department of Computer Science., 1994.

[12] Marco Bernardo. An algebra-based method to associate rewards with EMPA terms. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 358–368, Bologna, Italy, 1997. Springer-Verlag.

[13] Marco Bernardo, Lorenzo Donatiello, and Roberto Gorrieri. A formal approach to the integration of performance aspects in the modeling and analysis of concurrent systems. *Information and Computation*, 144(2):83–154, 1998.

[14] Marco Bernardo and Roberto Gorrieri. Extended Markovian process algebra. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 315–330, Pisa, Italy, 1996. Springer-Verlag.

[15] Marco Bernardo and Roberto Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1–2):1–54, 1998. Tutorial.

[16] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume I and II. Athena Scientific, 1995.

[17] A. Bianco and L. De Alfaro. Model checking of probabilistic and nondeterministic systems. *Lecture Notes in Computer Science*, 1026:499–513, 1995.

[18] Blackdown JDK Homepage. Hosted at http://www.blackdown.org/.

[19] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[20] Peter Buchholz. On a markovian process algebra. Technical Report 500, Universität Dortmund, Fachbereich Informatik, 1994.

[21] Michael J. Butler. Refinement and decomposition of value-passing action systems. In Eike Best, editor, *CONCUR'93 – 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 217–232. Springer-Verlag, 1993.

[22] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: a Foundation.* Addison-Wesley, Reading, Mass., 1988.

[23] Graham Clark. Formalising the specifications of rewards with PEPA. In *Proceedings of the Fourth Workshop on Process Algebra and Performance Modelling*, pages 139–160, 1996.

[24] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[25] Luca de Alfaro. Stochastic transition systems. In *Proceedings of CONCUR'98*, LNCS. Springer, 1998.

[26] Luca de Alfaro. From fairness to chance. *Electronic Notes in Theoretical Computer Science*, 22:33 pages, 1999.

[27] Debian GNU/Linux Homepage. http://www.debian.org/.

[28] Cyrus Derman. *Finite State Markovian Decision Processes*, volume 67 of *Mathematics in Science and Engineering.* Academic Press, 1970.

[29] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[30] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems.* Addison Wesley, 7th edition, 1995.

[31] Amani El-Rayes, Marta Kwiatkowska, and Steven Minton. Analysing performance of lift systems in PEPA. In *Proc. UK Performance Engineering Workshop*, page 17, September 1996.

[32] Jerzy Filar and Koos Vrieze. *Competitive Markov Decision Processes.* Springer, 1997.

[33] Clemens Fischer. Combining Z and CSP. Technical report, University of Oldenburg, 1996.

[34] B. L. Fox and D. M. Landi. An algorithm for identifying the ergodic subchains and transient states of a stochastic matrix. *Communications of the ACM*, 2:619–621, 1968.

[35] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[36] Paul Gardiner and Carroll Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993.

[37] S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In G. Haring and G. Kotsis, editors, *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794 of *LNCS*, pages 353–368. Springer-Verlag, 1994.

[38] David Gries. *The Science of Programming*. Springer, New York, 1981.

[39] Stefan Hallerstede. Semantische Fundierung von CSP-Z. Master's thesis, University of Oldenburg, Germany, 1997.

[40] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[41] Martin Hargreaves. *Engineering Systems: Modelling and Control*. Addison Wesley-Longman, 1996.

[42] Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent program. *ACM Transactions on Programming Languages and Systems*, 5(3):356–380, 1983.

[43] Boudewijn R. Haverkort. *Performance of Computer Communication Systems: A Model-based Approach*. John Wiley & Sons, 1998.

[44] Boudewijn R. Haverkort and Ignas C. Niemegeers. Performability modelling tools and techniques. *Performance Evaluation*, 25:17–40, 1996.

[45] Boudewijn R. Haverkort and Kishor S. Trivedi. Specification techniques for markov reward models. *Discrete Event Dynamic Systems: Theory and Applications*, 3:219–247, 1993.

[46] Jifeng He. Process simulation and refinement. *Formal Aspects of Computing*, 1:229–241, 1989.

[47] Jifeng He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP 86, European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer, 1986.

[48] Jifeng He, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2–3):171–192, 1997.

[49] Eric Hehner. Probabilistic predicative programming. Private Communication.

[50] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.

[51] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39:5–35, 2000.

[52] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A markov chain model checker. In *TACAS 2000*, number 1786 in Lecture Notes in Computer Science, 2000.

[53] Holger Hermanns and Michael Rettelbach. Syntax, semantics, equivalences, and axioms for MTIPP. In U. Herzog and M. Rettelbach, editors, *Proceedings of the Second Workshop on Process Algebra and Performance Modelling*, pages 71–88, 1994.

[54] Holger Hermanns, Michael Rettelbach, and Thorsten Weiss. Formal characterisation of immediate actions in SPA with nondeterministic branching. *The Computer Journal*, 38(7):530–541, 1995.

[55] Wim H. Hesselink. Nondeterminacy and recursion via stacks and games. *Theoretical Computer Science*, 124:273–295, 1994.

[56] Harro Heuser. *Lehrbuch der Analysis*, volume 1. B. G. Teubner Stuttgart, 10. edition, 1993.

[57] Jane Hillston. PEPA: Performance enhanced process algebra. Technical Report CSR-24-93, University of Edinburgh, Edinburgh, Scotland, 1993.

[58] Jane Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[59] C. A. R. Hoare. *Communcating Sequential Processes.* Prentice Hall, 1985.

[60] C. A. R. Hoare, Jifeng He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, 1987.

[61] Ronald A. Howard. *Dynamic Probabilistic Systems*, volume I: Markov Models. John Wiley and Sons, 1971.

[62] Ronald A. Howard. *Dynamic Probabilistic Systems*, volume II: Semimarkov Models and Decision Processes. John Wiley and Sons, 1971.

[63] JavaCC Homepage. Hosted at http://www.metamata.com/.

[64] Claire Jones. *Probabilistic Non-Determinism.* PhD thesis, LFCS, Edinburgh University, 1990.

[65] Bengt Jonsson, Chris Ho-Stuart, and Wang Yi. Testing and refinement for nondeterministic and probabilistic processes. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 418–430. Third International Symposium Organized Jointly with the Working Group Provably Correct Systems-ProCoS, Springer-Verlag, 1994.

[66] Bengt Jonsson and Kim Guldstrand Larsen. Specification and refinement of probabilistic processes. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science*, pages 266–277, 1991.

[67] Mark B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[68] Leslie Pack Kaelbling, Michael B. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[69] Krishna Kant. *Introduction to Computer System Performance Evaluation*. McGraw Hill, New York, 1992.

[70] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Undergraduate Texts in Mathematics. Springer-Verlag, 1976.

[71] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22, 1981.

[72] Ulrich Krengel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik*. Vieweg Verlag, 1988.

[73] Helko Lehmann. *On Reasoning about Action and Change in the Fluent Calculus*. PhD thesis, Declarative Systems and Software Engineering, ECS, University of Southampton, 2001.

[74] Christoph Lindemann. DSPNexpress: A software package for the efficient solution of deterministic and stochastic petri nets. *Performance Evaluation*, 22:3–21, 1995.

[75] Christoph Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. Wiley, 1998.

[76] Falko Lorenz. *Lineare Algebra*, volume I and II. B. I. Wissenschaftsverlag, 3. edition, 1992.

[77] Gavin Lowe. Representing nondeterminism and probabilistic behaviour in reactive processes. Technical Report PRG-TR-11-93, Programming Research Group, Oxford University, 1993.

[78] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159–195, 1996.

[79] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM transactions on computer systems*, 2:93–122, 1984.

[80] Rudolf Mathar and Dietmar Pfeifer. *Stochastic für Informatiker*. B. G. Teubner, 1990.

[81] A. K. McIver and Carroll Morgan. Demonic, angelic and unbounded probabilistic choices in sequential programs. Technical report, Programming Research Group, University of Oxford, 1998.

[82] Annabelle McIver, Carroll Morgan, and Elena Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. Technical Report TUCS-TR-173, TUCS - Turku Centre for Computer Science, April 22 1998. Wed, 22 Apr 1998 12:00:00 GMT.

[83] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[84] Isi Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998.

[85] Michael K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, C-31(9):913–917, September 1982.

[86] Carroll Morgan. Of wp and CSP. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 319–326. Springer-Verlag, 1990.

[87] Carroll Morgan. *Programming from Specifications*. Prentice Hall International, 2nd edition, 1992.

[88] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, 1996.

[89] Peter Morris. *Introduction to Game Theory*. Universitext. Springer, 1994.

[90] MySQL Homepage. http://www.mysql.com/.

[91] Mark Nelson and Jean-Loup Gailly. *The Data Compression Book*. M&T Books, second edition, 1996.

[92] Randolph Nelson. *Probability, Stochastic Processes, and Queueing Theory: The Mathematics of Computer Performance Modelling.* Springer, 1995.

[93] Norman S. Nise. *Control Systems Engineering.* Benjamin/Cummings, 2nd edition, 1995.

[94] Ernst-Rüdiger Olderog. *Nets, Terms and Formulas*, volume 23 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1991.

[95] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, 1996.

[96] Amir Pnueli and L. D. Zuck. Probabilistic verification. *Information and Computation*, 103:1–29, 1993.

[97] Amir Pnueli and Lenore Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1:53–72, 1986.

[98] Martin L. Puterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming.* Wiley Series in Probability and Mathematical Statistics. Wiley, 1994.

[99] Michael O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.

[100] Josyula R. Rao. Reasoning about probabilistic parallel programs. *ACM Transactions on Programming Languages and Systems*, 16(3):798–842, 1994.

[101] Wolfgang Reisig. *Petri Nets. An Introduction*, volume 3 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[102] Marina Ribaudo. On the aggregation techniques in stochastic petri nets and stochastic process algebras. *The Computer Journal*, 38(7), 1995.

[103] Marina Ribaudo. Stochastic petri net semantics for stochastic process algebras. In *Proc. 6th International Workshop on Petri Nets and Performance Models*, 1995.

[104] A. W. Roscoe. An alternative order for the failures model. Technical Mongraph PRG-67, Programming Research Group, Oxford University, 1988.

[105] A. W. Roscoe. Unbounded nondeterminism in CSP. Technical Mongraph PRG-67, Programming Research Group, Oxford University, 1988.

[106] Robin Sahner, Kishor S. Trivedi, and Antonio Puliafito. *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package.* Kluwer Academic Publishers, 1996.

[107] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. In *CONCUR'94*, volume 836 of *LNCS*, pages 481–496. Springer, 1994.

[108] Karen Seidel. Probabilistic communicating processes. *Theoretical Computer Science*, 152:219–249, 1995.

[109] Karen Seidel, Carroll Morgan, and Annabelle McIver. Probabilistic imperative programming: a rigorous approach. Technical report, University of Oxford, PRG, 1997.

[110] Linn I. Sennott. *Stochastic Dynamic Programming and the Control of Queueing Systems.* Wiley Series in Probability and Statistics. Wiley, 1999.

[111] Kaisa Sere and Elena Troubitsyna. Probabilities in action systems. In *Proc. of the 8th Nordic Workshop on Programming Theory*, 1996.

[112] J. M. Spivey. *The Z Notation.* Prentice-Hall International, 1989.

[113] Peter H. Starke. *Analyse von Petrinetz-Modellen.* Velag Teubner, 1990.

[114] Sun JDK Homepage. Hosted at http://java.sun.com/.

[115] Henk C. Tijms. *Stochastic Models: An Algorithmic Approach.* Wiley Series in Probability and Mathematical Statistics. Wiley, 1994.

[116] Elena Troubitsyna. Enhancing Dependability via Parameterized refinement. In *Proc. of Pacific Rim International Symposium on Dependable Computing*, 1999.

[117] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 327–338, 1985.

[118] P. Whittle. *Probability via Expectation.* Springer, 1992.

[119] J. C. P. Woodcock and Carroll Morgan. Refinement of state-based concurrent systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 340–351. Springer-Verlag, 1990.

[120] Jim Woodcock and Jim Davies. *Using Z.* Prentice Hall International, 1996.

[121] Wang Yi. Algebraic reasoning for real-time probabilistic processes with uncertain information. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 680–693. Springer-Verlag, 1994.