Stochastic Abstraction of Programs: Towards Performance-Driven Development

Michael James Andrew Smith

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2010

Abstract

Distributed computer systems are becoming increasingly prevalent, thanks to modern technology, and this leads to significant challenges for the software developers of these systems. In particular, in order to provide a certain service level agreement with users, the *performance* characteristics of the system are critical. However, developers today typically consider performance only in the later stages of development, when it may be too late to make major changes to the design. In this thesis, we propose a *performance-driven* approach to development — based around tool support that allows developers to use performance modelling techniques, while still working at the level of program code.

There are two central themes to the thesis. The first is to automatically relate performance models to program code. We define the Simple Imperative Remote Invocation Language (Siril), and provide a probabilistic semantics that interprets a program as a Markov chain. To make such an interpretation both computable and efficient, we develop an *abstract interpretation* of the semantics, from which we can derive a Performance Evaluation Process Algebra (PEPA) model of the system. This is based around abstracting the domain of variables to *truncated multivariate normal measures*.

The second theme of the thesis is to analyse large performance models by means of *compositional abstraction*. We use two abstraction techniques based on *aggregation* of states — abstract Markov chains, and stochastic bounds — and apply both of them compositionally to PEPA models. This allows us to model check properties in the three-valued Continuous Stochastic Logic (CSL), on abstracted models. We have implemented an extension to the Eclipse plug-in for PEPA, which provides a graphical interface for specifying which states in the model to aggregate, and for performing the model checking.

Acknowledgements

I would like to thank first and foremost my supervisor, Jane Hillston, for all the advice, support and feedback she has given me throughout my time at Edinburgh. I would also like to thank my second supervisor, Ian Stark, for his invaluable feedback and advice — in particular, with respect to Chapters 3 and 4. Of my other colleagues in Edinburgh, I am particularly grateful to Stephen Gilmore for many stimulating discussions, and to Mirco Tribastone for developing the Eclipse Plug-In for PEPA, which I used as a basis for one of my tools (see Chapter 7). I am also forever indebted to the immeasurable support from Flemming and Hanne Riis Nielson, and everybody else at the Language Based Technology group at DTU, during the final stages of writing this thesis — without them, this would not have been possible.

It has been a great pleasure to have met and discussed ideas with so many interesting people during the course of my PhD. I would like to thank everyone who has inspired and influenced my work, including (in no particular order) Herbert Wiklicky, Alessandra Di Pierro, Dave Parker, Gethin Norman, Tony Field and Jeremy Bradley. I also owe a debt of gratitude to Alan Mycroft for his advice and support when I was deciding whether to embark upon a PhD, and for pointing me in the direction of Edinburgh.

I could not have undertaken this research without the generous support from the Engineering and Physical Sciences Research Council during my first year, and from a Microsoft Research European Scholarship during my subsequent years. I would especially like to thank Microsoft Research and the Informatics Graduate School for providing the funding for me to attend numerous conferences.

I am grateful to my flatmates, the members of the Edinburgh Go Club, my classmates studying Japanese, and everyone else who I have come to know in Edinburgh, for their friendship, company and support — not forgetting my long-suffering office mates, who withstood a daily barrage of bad jokes. Last, but not least, I would like to thank my family and friends from Bridlington, for always being there for me. I hope that with this thesis I can play some small role in enhancing our understanding of Computer Science, and that I can give something back to everyone that has encouraged and supported me on this journey.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Michael James Andrew Smith*)

To my granddad, Ron Schofield (1931–2008)

Table of Contents

List of Figures

List of Notation

$\mathrm{ds}(C)$	Derivative set of component C, page 30
$\underset{L}{\bowtie}$	PEPA cooperation combinator over action types L, page 29
$\top$	Passive rate, page 28
a, b	Action types, page 28
C	PEPA component, page 28
r	Rate ($\in \mathbb{R}_{\geq 0}$), page 28
$r_a(C)$	Apparent rate of activity a in component C, page 29

$def(O.f)$	Definition of $O.f$, of the form $O.f(X,\ldots,X)\ \{C\}$, page 39
O	Object, page 39
$O.f$	Method f of object O, page 39
X	Variable, page 39
$X_{O.f}$	Return variable of the method $O.f$, page 39

(X, σ_X)	Measurable space, page 46
$\mathcal{M}$	The set of Lebesgue-measurable subsets of $\mathbb{R}$, page 47
μ	Measure, page 46
$\boldsymbol{B}(X, \sigma_X)$	The set of all measures on (X, σ_X), page 46

$\iota_{\boldsymbol{X}}(X)$	The index of variable X in $\boldsymbol{X}$, page 52
$\llbracket B \rrbracket$	The set of valuations that satisfy the condition B, page 49
$\llbracket C \rrbracket$	Deterministic semantics of a command C, page 48
$\llbracket O.f \mid C \rrbracket^S_{pa}$	Stages in the probabilistic automaton semantics of the command C in $O.f$, page 53
$\llbracket O.f \mid C \rrbracket^T_{pa}$	Transitions in the probabilistic automaton semantics of the command C in $O.f$, page 53
$\llbracket O.f \mid C \rrbracket_p$	Probabilistic semantics of a command C in method $O.f$, page 48
$\mathcal{F}(O)$	Set of methods for the object O, page 52

$\bullet_{O.f}$	Terminal stage of $O.f$, page 53
$\circ_{O.f}$	Start stage of $O.f$, page 53
e_q	Scaling of a measure by $q \in [0,1]$, page 49
$e_{[\![B]\!]}$	Truncation of a measure with respect to the condition B, page 49
$s[O.f]$	Internal stage with label $O.f$, page 53
$\boldsymbol{O}$	Vector of objects, of length M, page 52
$\boldsymbol{X}$	Vector of variables, of length N, page 52
$\boldsymbol{X}(i)$	The ith variable in $\boldsymbol{X}$, page 52

α	Abstraction function, page 69
$\bot, \top$	Unconstrained lower and upper bounds for intervals, page 72
γ	Concretisation function, page 69
$\leq_{\text{str}}$	Strict ordering on measures, page 70
$\mathcal{D}$	Concrete domain of measures, page 74
$\mathcal{D}^\sharp$	Abstract domain of measures, page 74
$\mu \vdash s$	State in the probabilistic interpretation, page 58
$L[\boldsymbol{B}, \boldsymbol{c}]$	Linear operator function, page 73
$R_E(O)$	External rate of the object O, page 62
$R_I(O)$	Internal rate of the object O, page 62
$T[\boldsymbol{a}, \boldsymbol{b}]$	Truncation function, page 72
$\boldsymbol{\mu}$	Mean vector, page 72
$\boldsymbol{\Sigma}$	Covariance matrix, page 72

$Vars(O.f)$	Variables of $O.f$, page 90
$Vars_{arg}(O.f)$	Argument variables of $O.f$, page 90
$Vars_{int}(O.f)$	Interface (argument and return) variables of $O.f$, page 90
$Vars_{loc}(O.f)$	Local variables of $O.f$, page 90
$\pi_{O.f}$	Projection of states and transitions onto the those of $O.f$, page 91
$\pi_{\boldsymbol{X}'}$	Projection of a measure onto $\boldsymbol{X}'$, page 90

$(S^\sharp, \alpha)$	Abstraction of a state space S, page 116
$\boldsymbol{P}$	Probability transition matrix, page 25
$\boldsymbol{Q}$	Infinitesimal generator matrix, page 111
λ	Uniformisation constant, page 112
$\mathcal{M}$	Markov chain, page 116
$\mathcal{M}^\sharp$	Abstract Markov chain, page 116

$\mathcal{M}^{\sharp\sharp}$	Abstract CTMC component, page 143
$\mathit{Embed}\mathcal{M}$	Embedded DTMC of a CTMC $\mathcal{M}$, page 112
$\mathit{Unif}_{\lambda}(\mathcal{M})$	Uniformisation of the CTMC $\mathcal{M}$ with uniformisation constant λ, page 112
$\overline{\boldsymbol{P}}$	Uniformised probability transition matrix, page 112
AP	Atomic propositions, page 25
L	Labelling function, page 25
r	Rate function, page 27
S	Concrete state space, page 25
$S^{\sharp}$	Abstract state space, page 115
$\boldsymbol{\pi}^{(0)}$	Initial probability distribution, page 25
$\boldsymbol{\pi}^{(\infty)}$	Steady state distribution, page 26

$\mathcal{P}_{\trianglelefteq}(\varphi)$	CSL path probability formula, page 113
$\mathcal{S}_{\trianglelefteq}(\Phi)$	CSL steady state probability formula, page 113
Φ	CSL state formula, page 113
$\Phi_1\ \mathcal{U}^I\ \Phi_2$	CSL time-bounded until formula, page 113
φ	CSL path formula, page 113
$X^I\ \Phi$	CSL time-bounded next formula, page 113

©	Context of a PEPA component, page 152
$\leq_{\text{rst}}^{B_{comp}}$	Context-bounded rate-wise stochastic ordering, page 153
$\leq_{\text{rst}}$	Rate-wise stochastic ordering, page 150
$\leq_{\text{st}}$	Strong stochastic ordering, page 124
B_{comp}	Comparative bound, page 153
B_{int}	Internal bound, page 153
S_{Φ}^{C}	Complement of a set $S_{\Phi} \subseteq S$, page 149
$\boldsymbol{P}(i,*)$	The ith row of matrix $\boldsymbol{P}$, page 125

$\oplus$	Kronecker sum, page 202
$\otimes$	Kronecker product, page 201
$\circledast$	Kronecker product of two PEPA components, page 135
$\odot$	Kronecker sum of two PEPA components, page 135
$\boldsymbol{I}_n$	$n \times n$ identity matrix, page 133

Chapter 1

Introduction

There is a real need, in the software development industry, for an engineered approach to building distributed systems with performance in mind. Although methodologies such as software performance engineering [164] are used in some particular application areas, they are not employed broadly across the industry. This is a real problem, since an increasingly large proportion of development projects are now distributed in nature. The only long term solution is to make performance modelling and analysis techniques more accessible to developers.

Until now, the application areas that have made the most use of performance analysis have predominantly offered performance critical services to other applications. For example, there has been a great deal of work concerning the performance of databases [136], storage networks [162], and low-level communication protocols such as TCP [172]. The problem comes when we look at higher-level distributed systems, such as web services and web applications, where we have system-specific performance requirements, but lack the resources — in terms of time, money and expertise — to undertake extensive performance analysis.

To address this problem, we need to consider how performance engineering can be made integral to the software development process, without requiring specialist expertise on the part of the developer. To this end, we propose in this thesis a framework for *performance-driven* development, as a supplement to existing software engineering methodologies. The aim is to provide integrated tool support that allows developers to obtain performance predictions for a software implementation — throughout the entire development process. These predictions can then be used to indicate whether or not the implementation must be altered in order to meet its performance requirements.

Before we proceed any further, let us take a step back and clarify exactly what we mean by ‘performance prediction.’ To the cautious reader, the idea of automatically predicting the performance of an implementation will immediately ring alarm bells — not least, because it seems to imply the ability to predict whether or not a program will terminate, which is undecidable. Needless to say, this is not what we mean. There are two important points to bear in mind when considering a performance prediction:

1. A performance prediction must be based on a number of *assumptions*, some of which are inherent in the modelling formalism used, and some of which need to be specified by the developer. It is not possible to say anything about the performance of a piece of code in isolation — factors such as the hardware used, the network conditions and topology, and the behaviour of the users will all have an impact on performance. The developer must make some assumptions about the context in order to obtain a performance prediction, and a performance prediction must only be considered within this context.

2. A performance prediction must be based on a *safe abstraction*. As we shall see momentarily, there are several stages of performance-driven development where abstraction is necessary to perform any sort of analysis at all. For example, even a small implementation can quickly become too large to analyse directly if we consider every possible value that an integer variable can take. Instead we need to perform our analysis at a more abstract level, but at the same time ensure that it is safe with respect to the performance predictions that it yields. More precisely, we must ensure that any prediction we give is correct — with respect to the assumptions made — at the expense of sometimes being unable to give an answer.

Starting with the source code of a distributed system, there are a number of possible ways to obtain performance predictions. The most obvious is to actually *execute* the code — either directly, or via a simulation — but this is of little use during the development process, when the code is only partially written. A more sophisticated approach is to use *performance modelling*, where we build and analyse a mathematical model of the system. The difficulty with this approach is that it requires significant skill and understanding of the system in order to build the model. Our proposal is to open up these techniques to developers by finding ways of automatically extracting performance models from code. This is one of the key challenges of performance-driven development — to relate performance models to program code, with partially written code corresponding to more abstract models.

The purpose of this thesis is to present this notion of performance-driven development, and to show that the approach is viable in theory. In doing so, we will build and expand upon existing techniques for program analysis and performance evaluation, thus developing a theoretical basis on which performance-driven development can take place. In order to make realistic progress in these areas, however, we have restricted our focus to one development language and one modelling language:

- The *Simple Imperative Remote Invocation Language* (Siril) — which we introduce in Chapter 3 — is a syntactically simple language, which captures the main features present in imperatively programmed distributed systems. We will use this as our development language, and the starting point of our analysis.

- The *Performance Evaluation Process Algebra* (PEPA) [91] is a widely-used language for compositional performance modelling, and is supported by a range of analysis tools. It is based semantically on continuous time Markov chains [108], which have a rich and well-studied mathematical theory that we can utilise. We will introduce PEPA in Section 2.5.2.

In the remainder of this chapter, we will describe the general setting of the thesis, and give an outline of our theoretical and practical contributions. We begin in Section 1.1 with a general description of performance-driven development, before discussing each stage of the process in more detail in Sections 1.2, 1.3 and 1.4. We will then conclude this chapter with an outline of the thesis, as well as identifying the main contributions that we have made, in Section 1.5.

1.1 Performance-Driven Development

To allow developers to reason about performance properties throughout the development cycle, we propose a new methodology called *Performance-Driven Development*. This is an iterative software development process, where we periodically generate a performance model from source code, analyse it, then use the results of the analysis to guide further development. The process, illustrated in Figure 1.1, consists of the following four stages:

1. *Abstraction* — producing a performance model from program code. This requires some information from the developer concerning their assumptions about the context in which the program will execute, and the parts of the program to

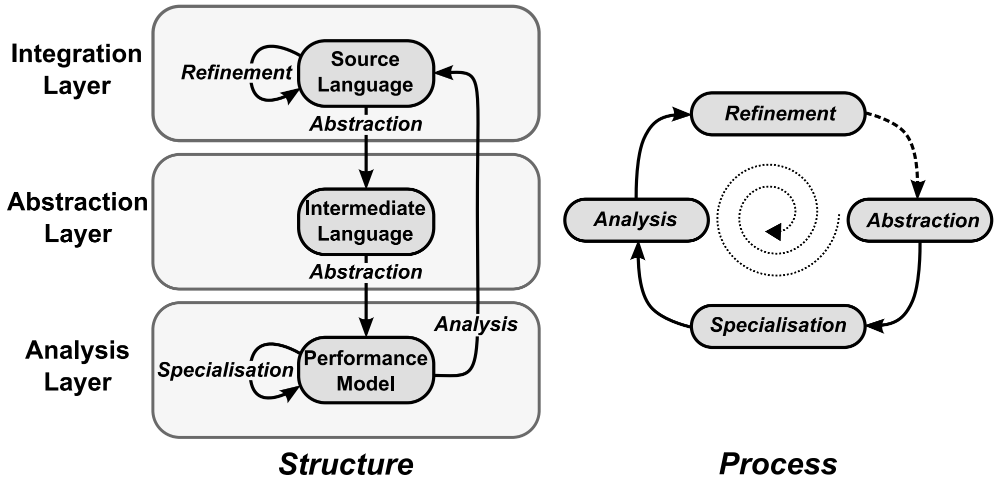


Figure 1.1: Overview of performance-driven development

explicitly model — for example, we will not usually want to model operating system calls. If we do not model part of the code, or if the code is incomplete — as is often the case early in development — we may need additional information from the developer about the expected functional behaviour and performance characteristics of the omitted code.

2. *Specialisation* — reducing the performance model so that it relates to a particular property of concern, and is small enough to analyse. This is a vital stage if we are to analyse systems of any realistic size. After abstracting the program code to a performance model, we will often find that it is too large to analyse, and so further abstraction is essential. If we are interested in a particular performance property, we can do this in such a way that the least important parts of the model (for that particular property) are abstracted the most.

3. *Analysis* — obtaining performance properties from the model. The analysis techniques depend on the particular modelling formalism used, but there are many existing tools available. For example, tool support for PEPA includes the PEPA plug-in for Eclipse [178], Möbius [48] and PRISM [114]. These support the numerical solution, model checking and simulation of PEPA models.

4. *Refinement* — using the results of the analysis to modify the implementation. It is the responsibility of the developer to modify their program code, but we still need to present the results of the analysis in a way that they can understand.

We will discuss the abstraction stage in more detail in Section 1.2, the specialisation stage in Section 1.3, and the analysis and refinement stages in Section 1.4.

In addition to these four stages, Figure 1.1 shows a structural view of performance-driven development, as a mapping between different languages. One interesting feature is that we identify an *intermediate language* between the source programming language and the performance modelling language. Since there is a large gap between the complexity of a real programming language, such as Java or C++, and that of an abstract performance model, separating the abstraction into two distinct stages helps to make it more manageable. An example of an intermediate language is Siril, which we mentioned in the introduction and will introduce in Chapter 3.

For the purposes of this thesis, we will only consider a fragment of the performance-driven development process, starting from this intermediate language Siril. That is to say, we will assume that any program we are interested in has already been translated into Siril, which allows us to focus on developing a formal and sound abstraction into a performance model, without having to deal with specific language features. Since our goal is to show that performance-driven development is a viable approach, it is important that we first lay down the theoretical foundations to support it. There remain, of course, a great many challenges in making it *practical*, but that is the topic of future work.

We will now examine each stage of performance-driven development in more detail, and in relation to the contributions of this thesis. In particular, this means that we will specifically talk about extracting PEPA models from Siril programs.

1.2 Abstraction: From Code to Model

The first stage of performance-driven development is to generate a performance model from program code. This is not straightforward, and we devote the entirety of Chapter 4 to one approach. First, however, let us consider this in a broader setting, by looking at the structure of the distributed systems we are interested in analysing.

We can think of a distributed program as consisting of a number of functional units, which each provide an external interface. If these functional units are placed in physically separate locations, then the only way for them to interact with one another is over a network. A user of the system — which may itself be another system — can interact either remotely, via the network, or locally, by directly invoking a function of the provided interface. In both cases, we can model each user as a separate parallel

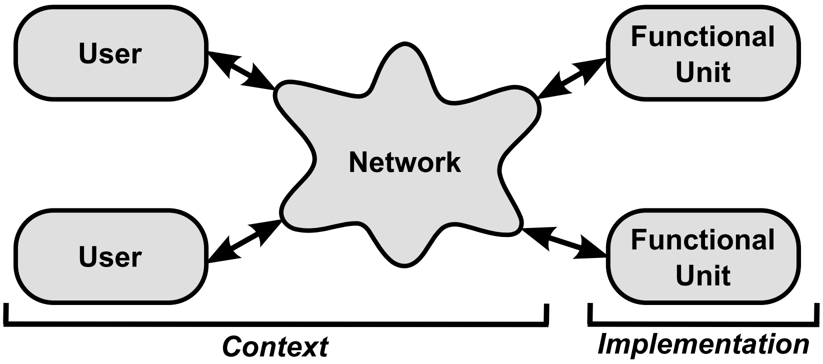


Figure 1.2: Structure of a distributed system

thread, with the characteristics of the user determined by the rate of interaction and the type of interaction (e.g. blocking or non-blocking). We can assume without loss of generality that all communication is via a network interface, since two threads on the same machine can communicate using the loopback interface. The different latencies can be captured by the timing parameters of the model.

The general layout of such a distributed system is shown in Figure 1.2. We can visualise a direct mapping between this structural view of the system and a PEPA model of its performance — the functional units, the network, and the users each correspond to *sequential components* in the model, and the way in which they interact is captured by the *system equation*. We will define these terms formally in the next chapter, and it will be useful to think back to this image when we do. Note that the network does not have to be a single component, and we could, for example, model a more complex network topology using multiple components that correspond to individual routers.

There are two aspects to generating a PEPA model from program code. The first is to abstract the functional units, which we do by analysing their code. The second is to specify the *context* — namely, the network and the users — which can either be done directly in PEPA, or by writing an additional functional unit in the source language, that can be abstracted to a PEPA component. In both cases, there is no need for the developer to be exposed to the modelling language — even if we describe the context directly in PEPA, we can provide a library of common network and user behaviours that can simply be 'plugged' into the model.

The greater challenge we face is abstracting the code of each functional unit to a PEPA component. We need to abstract both the possible data environments of the program — the values that its variables can take — and its control flow. In other words, we want to avoid having a state in the model for every possible valuation of a program's variables, and we want to avoid unrolling loops indefinitely. The problem arises because control flow decisions depend on the values of a program's variables, so we cannot abstract away too much information.

Our solution to this problem, which we present in Chapters 3 and 4, is to define a probabilistic semantics of our source language SIRIL, and build an *abstract interpretation* [50] on top of it. The main idea is to represent the state of the program as a probability measure, which evolves as the program executes. The result is that control flow decisions in the abstract model become *probabilistic*, since the abstract states represent a *distribution* over the values of the program's variables, rather than just one particular valuation. The focus of our analysis is to be able to calculate the probabilities of these control flow decisions, for distributed programs that interact using remote procedure calls. We will only consider loop-free programs, although we will discuss some of the challenges presented by loops in our conclusions.

This approach results in a probabilistic model of the program, but to reason about performance and construct a PEPA model we also need *timing information*. Fortunately, our abstraction ensures that every state in the abstract model corresponds to a basic block in the program — a portion of sequential code that does not contain any loops. We can therefore use profiling to measure the expected execution time of each block — if a block takes on average T time units to execute, we can say that it executes at a *rate* of $\frac{1}{T}$. By combining such timing information with the probabilistic model that we generate from the program, we can construct a PEPA model.

1.3 Specialisation of Performance Models

Once we have derived a PEPA model from a program, the next stage is to analyse it. The problem is that, in general, the model may be too large for the existing tools to analyse directly. We therefore need to look at ways of reducing the size of the model, which is the subject of Chapters 5 and 6.

Since PEPA is a compositional formalism, it allows extremely large Markov chains to be specified in a compact form. When we come to analyse a model, however, we need to actually generate the underlying Markov chain, which means that we lose this compact representation. There are many known techniques for reducing the size of Markov chains, which we will give an overview of in Section 2.5.3. However, since these techniques work at the level of the Markov chain, we run into problems when we have a PEPA model that describes a Markov chain that is simply too large to generate.

Our solution to this problem is based on two key ideas — we construct a *bounding* abstraction, and we construct it *compositionally*. The idea of a bounding abstraction is that properties of the Markov chain, which could be transient or steady state properties, are safely approximated by an *interval* of probabilities. For example, if the abstraction tells us that the steady state probability of being in a certain state is between 0.4 and 0.6, we will be sure that this is correct of the original model. Whilst a bounding abstraction is always safe in this respect, different abstractions can have different precisions — the challenge being to do better than the worst case answer that the probability lies between zero and one.

Our approach is to take two existing bounding abstraction techniques, and apply them compositionally to PEPA. We use abstract Markov chains [105] to bound transient properties of the model, and combine this with stochastic bounds [69] to bound steady state properties. The idea is to bound each sequential component in the model separately. This results in an abstract PEPA model, which induces a Markov chain that is a bounding abstraction of that induced by the original model. We will describe this in detail in Chapter 6.

The main challenge with this technique is in choosing an abstraction that gives the most precise bounds for a property of interest. Doing this automatically is a difficult and open problem, and one that we do not attempt to tackle in this thesis. We have instead implemented a graphical interface for specifying which states of a component to abstract, along with a model checker for the Continuous Stochastic Logic (CSL) [13, 16], as part of the PEPA plug-in for Eclipse [178]. We will describe this in Chapter 7.

1.4 Analysis and Refinement: From Model to Code

After obtaining a PEPA model that is small enough to analyse, we can use tools such as the PEPA plug-in for Eclipse to analyse the model. This allows us to obtain performance measures, such as the proportion of time spent in a particular state, or the throughput of a certain activity. The key difficulty is to present the results in a way that a developer can easily understand. This means that properties must be specified not in terms of the performance model, but in terms of the program code.

Some aspects of this translation are straightforward to do. For example, it is essential to keep a record of which statements of code each state in the performance model corresponds to, and which function call each named activity corresponds to.

The most difficult problem, however, is to find a more user-friendly language for *specifying* the performance properties, rather than having to use logics like CSL. Whilst this is an important practical issue, and there has been some recent work on addressing it [83, 175], it is beyond the scope of this thesis. We have, however, implemented a more user-friendly interface for constructing CSL formulae, which we will demonstrate in Chapter 7.

The aim of performance-driven development is to provide developers with performance information throughout the software development process. If the predicted performance does not meet the performance requirements, then the developer has the opportunity to modify the design at a much earlier stage than if they had waited until conventional performance testing. Whilst it is not possible in general to automatically re-factor an implementation so that it meets its requirements, performance analysis can help to identify a system's bottleneck components, and the reason for its failure.

It is vital, however, that these performance predictions are interpreted in the context of the assumptions made. If the developer assumes a more favourable network environment than in reality, they might incorrectly conclude that the system will meet its performance requirements. There are also a number of implicit assumptions in the modelling formalisms used, such as the use of exponential distributions in Markov chains. If we are to hide the details of performance modelling from developers, we need to be careful to make these assumptions known, so that they do not lead to false conclusions. Despite this note of caution, however, the benefits of bringing performance analysis techniques to developers are immense, as we hope will become apparent throughout this thesis.

1.5 Structure of the Thesis

An outline of the structure of the thesis is given in Figure 1.3, which shows how the chapters depend on one another. In Chapter 2, we describe the background to modern software development practices, and performance modelling techniques. We introduce PEPA, and discuss techniques for tackling the state space explosion problem in the context of Markovian performance models. We then move to the two main contributions of this thesis — Chapters 3 and 4 deal with abstracting programs to obtain performance models, and Chapters 5, 6, and 7 with specialising performance models so that they can be analysed. Each set of chapters can be read separately, but contribute equally significant roles in the development of performance-driven development as a

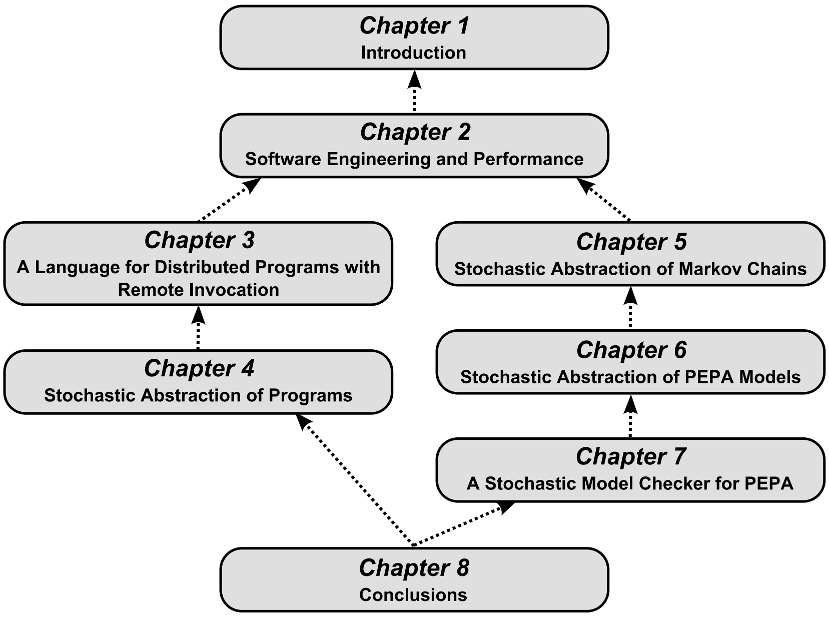

Chapter 1
Introduction
Chapter 2
Software Engineering and Performance
Chapter 3
A Language for Distributed Programs with Remote Invocation
Chapter 5
Stochastic Abstraction of Markov Chains
Chapter 4
Stochastic Abstraction of Programs
Chapter 6
Stochastic Abstraction of PEPA Models
Chapter 7
A Stochastic Model Checker for PEPA
Chapter 8
Conclusions

Figure 1.3: Outline of the thesis, and dependencies between chapters

viable approach.

In Chapter 3, we introduce the language Siril, and present a probabilistic semantics of Siril programs. We then construct an abstract interpretation of this semantics in Chapter 4, using truncated multivariate normal measures as the domain of the program's variables. We describe the abstract interpretation compositionally, and illustrate the approach by means of a running example.

Turning to the abstraction of PEPA models, we first describe the background to abstracting Markov chains in Chapter 5 — in particular, the techniques of abstract Markov chains and stochastic bounds. Building upon this work, we develop compositional abstractions for PEPA in Chapter 6, based on a Kronecker representation of PEPA models. We use abstract Markov chains to analyse transient properties, and stochastic bounds for steady state properties.

Bringing together the results in Chapter 6, we describe the tool support that we have developed in Chapter 7. We have implemented an extension to the PEPA plug-in for Eclipse, providing an abstraction engine for aggregating states in a PEPA model, and a CSL model checker for abstract PEPA models. Both are accessed using a graphical interface.

Finally, we conclude the thesis in Chapter 8 by examining our contributions and discussing some of the opportunities and challenges for extending this work further. In particular, we will discuss how we might deal with looping behaviour and more general models of communication in programming languages, in the context of constructing performance models. We will also examine ways in which we might further improve on techniques for abstracting compositional performance models. Ultimately, a great deal of future work is needed to bring performance-driven development close to being realised, but we hope that this thesis will provide the foundation for this to happen.

The main contributions of this thesis can be summarised as follows:

1. The idea of *performance-driven development* as a complementary approach to software development, based on tool support for deriving and analysing performance models from program code [Chapters 1 and 2].

2. A semantics of the *Simple Imperative Remote Invocation Language* (SIRIL) based on probabilistic automata (in this context, automata whose transitions are labelled with operators on measures) [Chapter 3].

3. An *abstract interpretation* of SIRIL programs, based on truncated multivariate normal measures, and a collecting semantics with which a PEPA model can be generated [Chapter 4].

4. A *compositional abstraction* of PEPA models, based on *abstract Markov chains*, for analysing transient properties of CSL/X[1] [Chapter 6].

5. A *compositional abstraction* of PEPA models, based on *stochastic bounds*, for analysing steady-state properties of CSL, and an algorithm for constructing these bounds over a partially-ordered state space [Chapter 6].

6. The development of *tool support* in the Eclipse platform for abstracting and model checking PEPA models, using the above techniques [Chapter 7].

Some of the work in this thesis has been previously presented. An earlier version of the work in Chapters 3 and 4 was published in [170], where the emphasis was on analysing looping behaviour rather than concurrency. The idea of extracting performance models from program code was first introduced in [169]. All other work, unless otherwise attributed, is original and unpublished.

[1] We do not include the timed next operator, for reasons we will discuss in Chapter 5.

Chapter 2

Software Engineering and Performance

We live, today, in a world where computing has become so integrated into our daily lives that we often fail to notice its presence. With the increasing pervasiveness and mobility of technology, the need for and the challenge of building robust distributed software systems is only set to increase. In addition to being free of errors, these systems must be able to handle the simultaneous demands of large numbers of users. The only hope to meet these requirements is through a systematic and disciplined approach to software engineering.

In comparison to other engineering disciplines, software engineering is still very much in its infancy. This is largely due to the fluid nature of software, which requires a fundamentally different approach to its development. Since software is an intellectual — rather than a physical — entity, it has the illusion of being financially cheap to modify. Issuing a patch for a piece of software is essentially free, compared with the cost of recalling a faulty car engine, or mobile phone. However, this frame of mind can easily lead to bloated systems that are too complex to understand, and therefore contain bugs that are impossible to fix.

As early as the 1970s, it was understood that a disciplined approach was needed to software development [60], and that completely sequential processes such as the waterfall model [151] were highly risky. This sort of approach requires the design to be fixed before starting the implementation, which among other things fails to account for any problems that might be discovered during coding. Even more worryingly, it assumes that testing only takes place after the implementation is complete, by which time it may be impossible to untangle and fix all the bugs. An analogy would be if one were to build a house, but only verify its structural integrity *after* construction work has been completed — even if the design is excellent, we need to adapt to unforseen circumstances, and detect mistakes as early as possible so that they can be remedied.

These early models of software engineering soon evolved into more realistic approaches, such as the spiral model [30]. This treats software development as an iterative process, involving several stages of risk assessment, planning, design and prototyping. In particular, by building testing into each cycle, we can identify and correct bugs at an earlier stage of development. More recently, the iterative approach has been taken to a further extreme with practices such as Extreme Programming (XP) [25]. This advocates having very short iteration cycles, as well as elevating the status of testing so that it *drives* the development process, rather than being driven by it.

This approach to testing, known as *test-driven development* [26], has had a major impact on the way in which software is developed. For example, tool support for unit testing is now widely available, which allows us to test individual components of a system in isolation. Unfortunately, testing at this level is usually concerned with only *functional* requirements; namely, that some method returns the correct value for a given input. *Non-functional* requirements, such as performance, often depend on the whole system and so are only tested at the system or integration level, which is usually fairly late in the development process. This essentially means that as far as performance testing is concerned, many industrial projects have not progressed much further than the waterfall model.

There are many techniques available for performance evaluation of software, but most development projects focus just on *measuring* the performance of a system in the late stages of testing. This involves subjecting the system to artificial workloads, which are designed to test its performance under both normal operating conditions and more extreme conditions. If it fails to meet its requirements, it may be possible to improve the performance by optimising and tuning bottleneck components and algorithms. But sometimes the problem is more serious. If the fault lies in the *design* of the system, it can be prohibitively expensive to fix in the late stages of development.

To try to mitigate these problems, there has been a large amount of research concerned with *modelling* performance. Even in the early stages of design, when there is no working system from which to measure performance, a performance model can be built to approximate its behaviour. Whether this be a mathematical model or a computer simulation, it allows performance problems to be identified at an earlier stage in development. This methodology, known generally as software performance engineering [164], provides a formal framework for including performance requirements in the design phase. Some application areas, such as low level communication protocols, processor scheduling algorithms, and traffic management protocols, have made use of performance modelling to great success [115]. However, the majority of commercial development projects still fail to make use of these techniques.

There are many reasons for not including performance modelling as part of a development project. Building and analysing mathematical models requires specialist knowledge and experience, and simulations are time consuming and costly to develop. The application domains that *do* use performance modelling often provide a low-level service with performance-critical algorithms. However, even for higher level applications, the way in which functionality is distributed over the network — a fundamental part of the design — is crucial to achieving good performance.

If developers are to use performance modelling in general, they require tools that are easy to use and that automate as much of the modelling process as possible. The topic of this thesis is to present one approach that might help this to become a reality — namely, the automatic extraction of performance models from source code. In this thesis, we put forward the idea of a new approach to *performance-driven* software development, but first we will examine the current state of affairs in more detail.

In this chapter we will provide a survey of the existing work on performance analysis of software, from both a software engineering and a performance modelling perspective. We begin in Section 2.1 with the main approaches to modern software development, along with techniques for testing and verification in Section 2.2. Following this, we discuss how performance fits into the development process in Section 2.3, before looking at methods for measuring and modelling the performance characteristics of a system in Sections 2.4 and 2.5 respectively. Finally, in Section 2.6, we describe existing work on extracting performance models from software specifications, before considering the main idea of this thesis — extracting performance models from source code — in Section 2.7.

2.1 Modern Software Development

There is no single approach to software engineering that can be definitively called "The Software Development Process." The wide range of techniques used in practice, however, can be thought of as lying on a scale between *predictive* and *adaptive* methods [29]. On the predictive end of the scale is up-front design [171], where the design

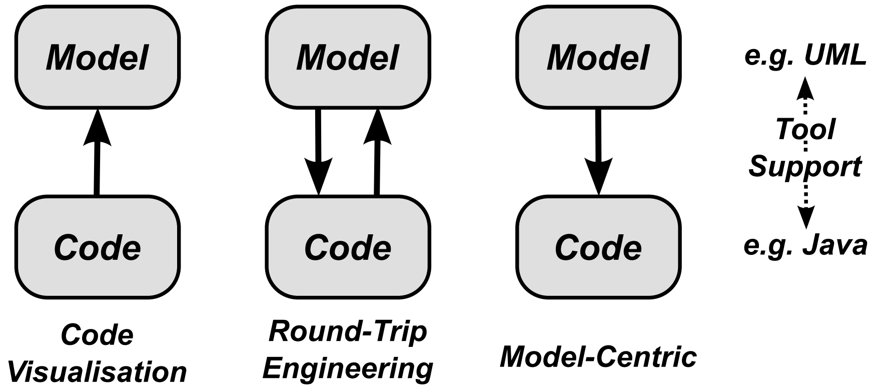


Figure 2.1: Relationships between model and code in software development

of the system mostly takes place before implementation starts. This commonly results in large design documents, which often use formalisms such as the Unified Modelling Language (UML) [152].

In contrast, adaptive methods are characterised by agile development [121], where the software is frequently inspected, and is subsequently adapted alongside the design. The very nature of this process encourages the use of light-weight design documents, and in many cases the only specification of a component is a suite of tests. This has the advantage that any structural changes in the code must be reflected in the tests, and vice versa, hence the code and specification remain synchronised.

One of the major problems with up-front design is that it is easy for the specification to become out-dated as the implementation evolves. When changes are not reflected in the specification documents, they not only become useless as an aid to understanding the system, but any verification that took place at the design level can no longer be trusted. For example, if we verify that a design satisfies some performance property, but then make modifications when we come to implementing it, we have no guarantee that the property will still hold of the system.

To address this problem, *model-driven development* [181] has been proposed as a methodology. The idea is to view the *model*, rather than the code, as the primary object of development. Fundamental to the approach is that we have a formal modelling language, such as UML, and that there is a mapping — ideally with tool support — between code and model. Figure 2.1 illustrates the different ways in which the code and model can interact. The ultimate aim is to perform round-trip engineering [85, 125], whereby changes in the code are reflected in the model and vice versa. This allows developers to alternate between updating the model and the code. Whilst the model-driven approach is most closely associated with up-front design, it is not incompatible with agile methodologies. Indeed, there are hybrid approaches, such as agile model-driven development [10], where both modelling and test-driven development take place in short, iterative cycles.

As far as commercial development goes, there has been a shift in recent years towards agile methodologies, particularly within the scope of interactive and web-based systems, where requirements are difficult to capture up-front. However, in a survey of software developers in 2003, it was found that 30% of respondents were still using the waterfall development model [132]. This evidence suggests that both adaptive *and* predictive methods are still widely used throughout the industry.

Key to the success of any development practice is the tool support available. Model-driven development requires tools to automatically generate skeleton code from a model, and also to update the model based on changes made to the code. Test-driven development requires an integrated and easy-to-use testing platform for the programming language being used. Furthermore, with the rise in integrated development environments such as Eclipse [1] and Visual Studio [4], it is important for the uptake of a tool that it integrates with these platforms. For a *performance-driven* approach to development to be used in practice, it is therefore vital that we also develop tool support.

2.2 Testing and Verification

An essential part of development is to ensure that an implementation is correct with respect to a specification — in other words, that when we write a program, it does what we want it to do. Ideally, we would like to *verify* that this is the case, to be sure that the program is correct for all inputs. In general, this cannot be done by a brute-force search, since the number of possible inputs is much too large. Instead, we need to build an *abstraction* of the program, with a state space that is small enough to verify. The central idea is that the abstraction must be 'safe' — if the abstraction satisfies a given property in the specification then we can be sure that the original program also satisfies it, but if the abstraction does not, we cannot infer anything about the program.

In most commercial development projects, bar certain safety critical systems such as aircraft control, formal verification is not used due to its expense. Instead, developers accept that they cannot verify a program for all possible inputs, and instead construct a set of *tests* for particular inputs. These tests should have a high *coverage* [186]

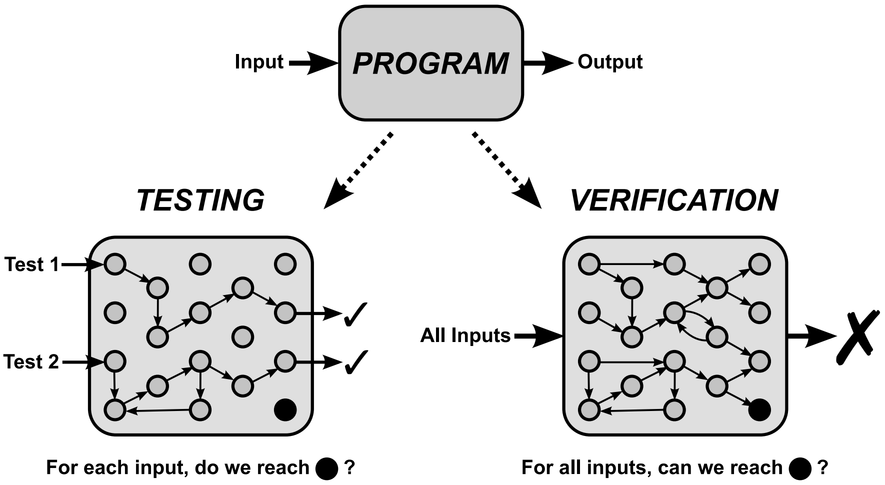

Input
PROGRAM
Output
TESTING
VERIFICATION
Test 1
Test 2
✓
✓
All Inputs
✗
For each input, do we reach ● ?
For all inputs, can we reach ● ?

Figure 2.2: Testing versus verification of programs

of the code, in that they test as many different execution paths as possible. There are different levels of software testing — *unit testing* looks at the functionality of an individual component (such as a Java class), *integration testing* looks at the interaction between components, and *system testing* looks at the entire system as a whole.

The difference between testing and verification is illustrated in Figure 2.2. The black state indicates a violation of the specification, where the incorrect output is given for some input. This violation may or may not be discovered during testing, depending on the coverage of the tests. Formal verification will at worst conclude that there *might* be a violation of the property, and at best that there *is*. However, unless we can automatically abstract and verify a program, it is too time-consuming for most developers.

The most widely used example of automatic verification is a type checker [139]. By abstracting away from the actual values that a variable can hold, and only considering the *types* of value, we can safely verify that, for example, a program never attempts to store a floating point value in an integer variable. Further forms of abstraction and verification, such as data flow analysis [12], are used internally by a compiler, in order to safely apply optimisations. These techniques are related in that they all analyse program code without actually running the program — they are collectively known as *static analysis* [133].

An alternative approach to static analysis, particularly in the context of distributed systems, is *model checking* [99]. The basic premise is to build an abstract model of the system in question, in an abstract modelling language. One such example is Promela,

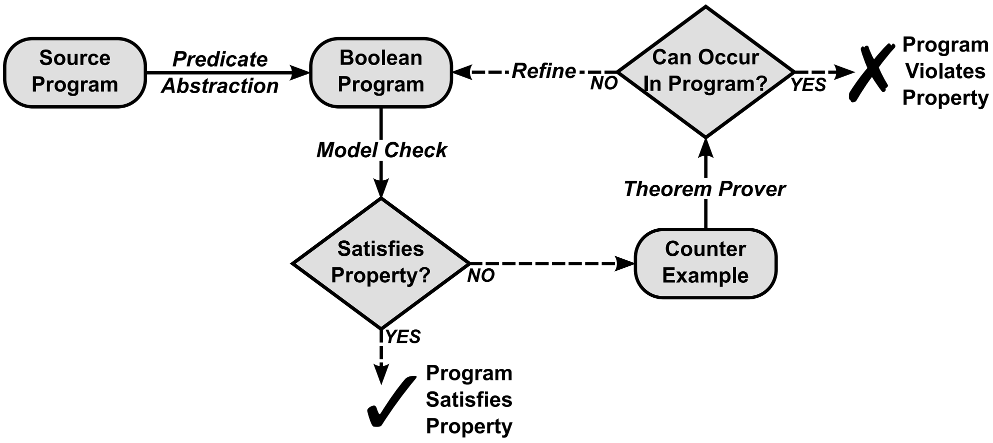


Figure 2.3: Counterexample-guided refinement of source code

the language of the widely used SPIN model checker [97]. Properties of the model are described in a temporal logic, and verification essentially becomes a question of reachability — whether it is possible to reach a state that violates the property.

Since model checking requires significant skill on the part of the developer, there has been a drive to look for ways to automate this process — namely, how to model check real code, rather than an abstract modelling language. A recent success story is the work on SLAM [22] and Blast [86], which has led to Microsoft's static driver verifier [21]. This is a tool that has been made freely available to Windows device driver developers, allowing them to verify that their code conforms to the API. To give an example of how error-prone device driver programming is, the tool discovered a subtle bug in the standard parallel port driver, which has been available as part of the driver development kit for many years, and undergone extensive testing.

Figure 2.3 shows how this process works. Firstly, the program (in C) is converted into a Boolean program, by a process called *predicate abstraction*. This makes all control flow decisions non-deterministic, and uses Boolean predicates in place of program variables, so that the state space is small enough to model check. Of course, this abstraction allows more possible behaviours than the original program, so if it fails to satisfy the property, this does not mean that the original program also fails. In this case, the model checker generates a *counter-example* to the property — namely, a way to reach a state that violates the property. If this counter-example cannot occur in the original program, the abstract program is then *refined* by adding an additional predicate. This process continues until an actual counter-example is found, or else the abstract program satisfies the property; in which case, so does the original program.

There has been other work concerned with model checking source code directly. One example is the C Bounded Model Checker (CBMC) [46], which uses bounded model checking to verify properties of ANSI-C programs. This essentially means that loops are unrolled only up to a certain limit, after which the possibility of re-entering the loop is ignored. The program is then converted into static single assignment form, and expressed as a logic formula C. We can then verify a property P by determining whether $C \wedge \neg P$ is satisfiable — if it is, then the property is false.

A similar approach, applied to a higher level language, is the extended static checker for Java (ESC/Java) [66]. This relies on user annotations in the Java Modelling language (JML) [116], which assert properties that must hold at particular points in the code. The tool translates these assertions, and the code itself, into an intermediate language of guarded commands, from which it can generate a set of verification conditions. These are then passed to a theorem prover for verification. Other tools that work at the language level aim to make verification and code maintenance easier for developers. CCured [131], for example, allows legacy C code to be made type-safe, and for various other analyses to be carried out. It makes use of the C Intermediate Language (CIL) [130], which is an intermediate form of C, without its complicated and ambiguous constructs.

We can take great heart from these successful applications of software verification techniques at the level of program code. This is the sort of approach that performance evaluation needs to take if it is to become more widespread throughout the development community — we will discuss this further in Section 2.7.

2.3 Performance in the Development Process

We have so far discussed software development practices, and the testing and verification of software, with little reference to performance. Current industrial practice regarding the specification, testing and verification of performance requirements is far behind that of functional requirements. This is not to say, however, that good performance practice does not exist in industry, and there are many application areas where software performance engineering [167, 165] is successfully being applied.

It is important not to confuse *performance engineering* with *performance tuning*. The former is concerned with designing and building a system that meets certain performance criteria, whilst the latter is concerned with optimising an existing system, to meet these criteria. Most development projects of any reasonable size undergo performance tuning in order to improve efficiency, but relatively few undertake disciplined performance engineering, starting in the design phase.

This comes as quite a surprise, when we consider that mathematical models such as queueing networks [110] were already being used in the 1960s in the context of time-shared operating systems [109, 156]. In the 1970s, attempts were made to formalise the design and specification of software performance criteria [80, 161], but this failed to make its way into industrial development practice. In spite of this, advances in mathematical modelling techniques continued to be made, and software performance engineering began to emerge as a discipline in its own right [165].

The first stage in performance engineering is performance requirements capture, which is typically done by constructing *use cases* — particular scenarios of the user interacting with the system. Each use case will contain performance criteria — such as the user receiving a response within a given time period — and therefore describes a performance scenario. To verify that the design can satisfy this scenario, a *performance model* needs to be built and analysed. There are many different modelling formalisms that can be used to do this, and we will discuss them in Section 2.5.

It is important that performance engineering be viewed as a complementary approach, and not a replacement, for other software engineering methodologies. An illustration of how it can take place in parallel with the usual stages of software development is shown in [71], and more recently there have been extensions to UML that allow performance specification alongside behavioural specification [81, 82]. There has also been some study of software performance *anti-patterns* [166], the aim of which is to provide a set of design heuristics with respect to performance. A performance anti-pattern is a design feature that tends to result in poor performance, and should not generally be used. In other words, it highlights a common design mistake.

Although it is better to design a system with performance in mind, rather than hope to fix it later, *performance tuning* is still a vitally important aspect of modern software development. Even for well-engineered systems, it is considered bad practice to optimise too soon at the algorithmic level, since 'cleverer' code is more likely to contain bugs. To quote Knuth, "we should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" [111]. For this reason, most development projects undergo *performance testing* [180] as part of the system level testing, to identify where optimisation is required.

There are a number of testing strategies that fall under the umbrella of performance testing. *Load testing* measures the system subject to a particular workload — often a characteristic, or *benchmark* workload. These measurements are then compared to the performance specification, to determine whether optimisation is needed. *Endurance testing* ensures that the system can maintain its performance over a sustained period of time, with the aim of detecting any memory leaks. *Stress testing* pushes the system beyond its performance limits, to verify that it fails in a safe manner, and subsequently recovers.

Whilst performance testing is usually undertaken *after* integration testing — towards the end of the development process — it can also take place much earlier. One approach is to take the code that has already been implemented, and combine it with *stubs*, generated from the use case that we want to test [58]. Performance testing can also be carried out at the level of unit testing, where we examine the performance properties of individual components. For example, JUnitPerf [2] is an extension to the unit testing framework JUnit, which allows load testing of Java classes. Whilst this approach is hard to scale to system level performance properties, it can enable component level performance bugs to be found at an early stage in development.

2.4 Measuring Performance

The key to performance testing is being able to observe and measure the behaviour of a system. There are fundamentally two ways to approach this. Either we consider the system as a black box, and measure performance information from the perspective of the user (or an external system), or we analyse the code itself, to determine how much time is spent in individual methods. The latter is known as *profiling*, and can be used to identify the bottleneck components of the system — the most important parts to optimise. This is probably the most widely used technique for performance measurement in practice.

If we have support from the underlying virtual machine or interpreter, we can use *event-based* profiling. Usually, an API provides hooks to the profiler, to inform it of events such as calling a method or raising an exception. If such an API is not available, we can use *statistical* profiling, where we periodically poll the current program counter (by way of an operating system interrupt), and use this to build up an image of where the program spends most of its time. An alternative approach is to first *instrument* the program code, by adding instructions that explicitly calculate durations. This has the disadvantage, however, that it might alter the behaviour of the program, introducing so-called ‘heisenbugs’. In addition to identifying which parts of the program we spend the most time in — the bottleneck components — profiling can be used in conjunction with static analysis to predict the average execution time, and its variance [153].

Although profiling is very useful for guiding the performance tuning of software, we are often more interested in performance from the perspective of the user. In this case, we want to measure observable performance criteria, such as response time or throughput. Operational analysis [59] uses simple mathematical laws to derive performance properties from observable data. For example, we can determine the bottleneck components in a system, without requiring an elaborate measurement infrastructure.

In the compiler community, there is considerable interest in *branch prediction* — namely, predicting whether or not a branch in the code will be taken [168]. This is related to profiling, in that modern superscalar processors store a cache of whether or not each branch has been taken in the past, up to some history — essentially, profiling the program. The past behaviour of the branch is used to predict its future behaviour, and a correct prediction will maximise the number of useful instructions that the processor executes [160]. There has also been interest in static branch prediction, whereby the probability of taking a branch is determined at compile time. For example, value range propagation [137] determines the range of values that a variable can possibly hold, in order to predict the probability of a branch being taken.

In the context of large scale distributed systems, particularly those involving server farms or grid computing [67], performance measurements are quite frequently used to predict the future behaviour of the system. The general approach is to use models to predict where a workload should be distributed in order to minimise response time, power consumption, or some other property. For example, the work in [14] attempts to predict the response time of a workload on a different server architecture to the one currently being used. When the server is very busy, a workload can then be shipped to a new server and still meet its performance contract.

2.5 Modelling Performance

Whilst measurement techniques are important for performance testing and dynamic adaptability, if we want to evaluate the performance of a system before it has been built, we have no choice but to build a *model*. This is necessarily an abstraction, based on the design of the system, that captures its key performance criteria — this could be for the whole system, or for a particular use case. There are generally two approaches we can take to modelling — either we build a custom *simulator* of the system, or we build a *mathematical model*. We shall focus our attention on the latter.

Usually, when modelling a complex system, we cannot be certain about the duration of some events. For example, the time taken for a packet to travel across the network will vary, depending on the amount of congestion and the particular route taken. Hence, rather than being deterministic, durations are usually modelled *stochastically*. There are many different stochastic formalisms, including queueing networks [110], layered queueing networks [73], stochastic automata networks (SAN) [142], stochastic Petri nets [20, 120, 127], and stochastic process algebras [28, 31, 87, 91, 145].

Most of these formalisms have a semantics based on the *Markov chain* [108], which we will introduce formally in the next section. The key concept is the *memorylessness property*, which means that the behaviour of the model depends only on its current state, and not on how it arrived at that state. This condition inevitably restricts the type of distribution that can be used to describe the duration of events — specifically to geometric or exponential distributions, depending on whether we use a discrete or continuous notion of time. This allows for efficient numerical analysis techniques, so that we can compute performance measures of the model. There exist more general formalisms that relax this restriction, such as generalised semi-Markov processes [123], but this makes numerical analysis much more difficult.

In addition to stochastic models, other formalisms are used to describe different types of timed system. Timed automata [9] are important for analysing systems with hard time constraints, such as real-time embedded systems. A timed automaton is essentially a non-deterministic automaton with a clock, whose transitions are labelled with constraints on the value of the clock — for example, requiring that we leave a state between 5 and 10 seconds after entering it. Probabilistic timed automata [101] allow us to include stochastic behaviour in such formalisms. In the context of control theory, where systems have both a discrete and a continuous element to their state space, hybrid automata [8] have been developed. These can be thought of as an extension of timed automata, where clocks can represent quantities other than time — for example, temperature — and can evolve over time according to an arbitrary set of differential equations.

Throughout this thesis we will restrict our attention to stochastic models over an entirely discrete state space, and we will present the basic notions that we build upon in the remained of this section. In Section 2.5.1 we will introduce the basic notions of discrete and continuous time Markov chains, before introducing the Performance Evaluation Process Algebra (PEPA) in as a higher-level modelling language, in Section 2.5.2. We will then discuss techniques for dealing with the state space explosion problem, in Section 2.5.3.

2.5.1 Markov Chains

A *Discrete Time Markov Chain* (DTMC) is essentially a state machine, whose transitions are labelled with probabilities. It describes a dynamic system in which the state evolves over time — but in discrete time steps. If we are in a state s at one moment in time, then the model will move into state s' at the next moment in time with a certain probability — given by the label p on the transition $s \xrightarrow{p} s'$. This means that at any moment in time, the state of the DTMC is given by a *probability distribution* over its state space.

The formal definition of a DTMC is as follows:

Definition 2.5.1. *A* Discrete Time Markov Chain (DTMC) *is a tuple* $(S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, L)$*, where* S *is a countable and non-empty set of states,* $\boldsymbol{\pi}^{(0)} : S \to [0,1]$ *is an initial probability distribution over the states,* $\boldsymbol{P} : S \times S \to [0,1]$ *is a function describing the probability of transitioning between two states, and* $L : S \times AP \to \{\mathtt{tt}, \mathtt{ff}\}$ *is a labelling function over a finite set of propositions* AP*.*

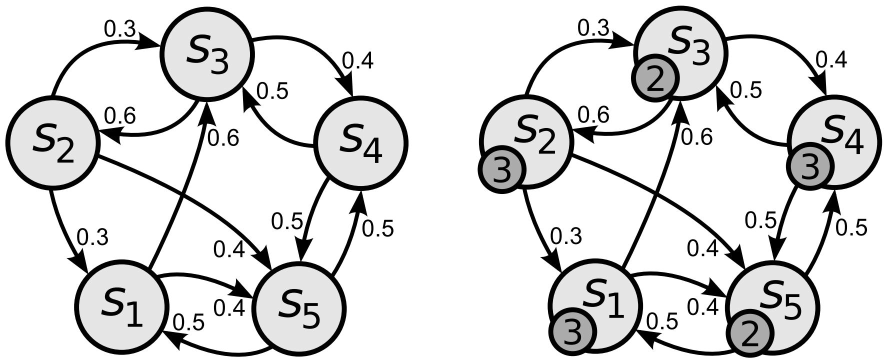
Note that in order for $\boldsymbol{\pi}^{(0)}$ to describe a probability distribution, it must be the case that $\sum_{s \in S} \boldsymbol{\pi}^{(0)}(s) = 1$, and similarly, for $\boldsymbol{P}$ to describe a probability distribution over the choice of transition, we require that for all $s \in S$, $\sum_{s' \in S} \boldsymbol{P}(s, s') = 1$.

The function $\boldsymbol{P}$ describes the probability $\boldsymbol{P}(s_1, s_2)$ of transitioning between two states s_1 and s_2 of the Markov chain in a single time step. The duration of this time step is not specified, in the sense that we describe only changes in the state space, and not how long they take. As the DTMC evolves, we can describe a probability distribution[1] $\boldsymbol{\pi}^{(t)}$ over its states at each point in time t — we this the *transient distribution* at time t. For a DTMC $(S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, L)$, and a distribution $\boldsymbol{\pi}^{(t)}$, we can compute $\boldsymbol{\pi}^{(t+1)}$ as follows:

$$\boldsymbol{\pi}^{(t+1)}(s') = \sum_{s \in S} \boldsymbol{\pi}^{(t)}(s) \boldsymbol{P}(s, s')$$

Figure 2.4(a) shows a graphical representation of a DTMC with five states: its state space $S = \{s_1, s_2, s_3, s_4, s_5\}$. If we take an initial distribution $\boldsymbol{\pi}^{(0)}$ such that $\boldsymbol{\pi}^{(0)}(s_1) = 1$,

[1] Note that the traditional mathematical definition of a DTMC is in terms of a sequence of random variables, indexed by time: $X_0, X_1, X_2, \ldots$ — if we take the codomain of the random variables to be $(S, \mathcal{P}(S))$, then $\Pr(X_t \in \{s\}) = \boldsymbol{\pi}^{(t)}(s)$.



(a) Discrete Time Markov Chain **(b) Continuous Time Markov Chain**

Figure 2.4: Discrete and continuous time Markov chains

then we can iteratively compute the distribution for each point in time, using the above equation:

$$\begin{array}{lcl ccccc l}
 & & & s_1 & s_2 & s_3 & s_4 & s_5 & \\
\boldsymbol{\pi}^{(0)} & = & [& 1.0000, & 0.0000, & 0.0000, & 0.0000, & 0.0000 &] \\
\boldsymbol{\pi}^{(1)} & = & [& 0.0000, & 0.3000, & 0.0000, & 0.4000, & 0.3000 &] \\
\boldsymbol{\pi}^{(2)} & = & [& 0.1800, & 0.1800, & 0.3200, & 0.1200, & 0.2000 &] \\
\boldsymbol{\pi}^{(3)} & = & [& 0.1080, & 0.3340, & 0.1320, & 0.3120, & 0.1140 &] \\
\boldsymbol{\pi}^{(4)} & = & [& 0.2004, & 0.1668, & 0.2896, & 0.1548, & 0.1884 &] \\
\vdots & & & & & & & & \\
\boldsymbol{\pi}^{(\infty)} & = & [& 0.1484, & 0.2474, & 0.2141, & 0.2303, & 0.1567 &]
\end{array}$$

Here, $\boldsymbol{\pi}^{(\infty)} = \lim_{t\to\infty} \boldsymbol{\pi}^{(t)}$ is called the *steady state distribution*. This limit exists for any initial distribution, if the DTMC is *ergodic*. If the state space is finite, this requires the DTMC to be *irreducible* — every state is reachable from every other state — and *aperiodic* — it is not the case that any state can only be reached in multiples of k time steps, for $k > 2$. Note that as an alternative to computing the above limit, we can calculate $\boldsymbol{\pi}^{(\infty)}$ by solving the following set of linear equations:

$$\boldsymbol{\pi}^{(\infty)}(s') = \sum_{s\in S} \boldsymbol{\pi}^{(\infty)}(s)\boldsymbol{P}(s,s')$$

While a DTMC describes a system that evolves probabilistically at discrete moments in time, it does not allow us to talk about the *duration* of each time step. We can introduce a notion of time by associating with each state s in the Markov chain a random variable $X(s)$ that gives the *sojourn time* for the state — in other words, how long we remain in the state before leaving it. If this is exponentially distributed with a rate parameter

$r(s)$, such that $\mathbf{Pr}(X(s) < t) = 1 - e^{-r(s)t}$, then we have a *Continuous Time Markov Chain (CTMC)*. This is defined formally as follows:

Definition 2.5.2. *A* Continuous Time Markov Chain (CTMC) *is a tuple* $(S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$*, where* S*,* $\boldsymbol{\pi}^{(0)}$*,* $\boldsymbol{P}$ *and* L *are defined as for a DTMC, and* $r : S \rightarrow \mathbb{R}_{\geq 0}$ *assigns an exit rate to each state. If* $r(s) = 0$ *then no transitions are possible from state* s*, and we require that* $\boldsymbol{P}(s,s) = 1$*, and* $\boldsymbol{P}(s,s') = 0$ *for all* $s' \neq s$*.*

By this definition, we allow the possibility of an exit rate of zero from a state. We interpret this as an absorbing state, by stating that the probability of leaving the state is zero, which ensures that for every state s, $\sum_{s' \in S} \boldsymbol{P}(s,s') = 1$. This is an unusual definition, but it will be useful later in the thesis, when we look at compositional representations of Markov chains (see Chapter 6).

Figure 2.4(b) shows a graphical representation of a CTMC — where we add an exit rate to each state of the DTMC in Figure 2.4(a). As with a DTMC, we can compute the steady state distrubution of a CTMC if it is ergodic. In this case, for a finite state space, we require only that the CTMC is irreducible — periodicity is not possible, since time is continuous. There are a number of ways to analyse both CTMCs and DTMCs — either by numerical performance evaluation, model checking, or stochastic simulation [77, 150]. We will look at *stochastic model checking*, and the use of *logics* for property specification, in Chapter 5.

The formalisms we have introduced in this section are purely stochastic, in the sense that the models they describe have no *non-determinism* in their behaviour. A formalism that combines stochasticity with non-determinism is a *Markov Decision Processes (MDP)* [146] — which can be either discrete or continuous time, as with Markov chains. In a discrete time MDP, rather than making a probabilistic choice as to the next state in the model, given that we are in a particular state, we make a *non-deterministic choice* between several probability distributions. If we supply a strategy for resolving the non-determinsim — known as a *scheduler* — then we end up with a DTMC. Hence, we can view a discrete time MDP as describing a *set* of DTMCs.

Philosophically, there are two ways to interpret the non-determinism in an MDP. One view is that the non-determinism describes part of the system that we cannot control — for example, a user of the system. It is therefore an *intrinsic* part of the model, since we want to account for all possible ways of interacting with the system. The other view is that non-determinism is an *abstraction*, since it represents uncertainty about the behaviour of the system — in other words, the system can really be described by a stochastic model, but we do not know which probabilities to use.

In the context of this thesis, we will primarily take the latter interpretation, where we use non-determinism as an abstraction. We will introduce this idea more formally in Chapter 5, when we look at abstractions of Markov chains.

2.5.2 The Performance Evaluation Process Algebra

Markov chains are an important mathematical formalism for performance modelling, however working with them directly is cumbersome, particularly for very large models. For example, if we want to build a model with a million states, we could not expect to explicitly enumerate every state and every transition, without making mistakes. Mathematicians typically avoid this by describing the state space *parametrically*, but this approach is not compositional, and requires a detailed understanding of the model at the system level. Computer scientists, on the other hand, use high-level *modelling languages* to describe models *compositionally* — namely, modelling a system in terms of smaller components. We can then compile the model into an underlying mathematical formalism, such as a Markov chain.

In this thesis, we will take a particular focus on one modelling language — the Performance Evaluation Process Algebra (PEPA) [91]. PEPA is a stochastic process algebra, which can be used to compositionally describe a CTMC. In PEPA, a *system* is a set of concurrent *components*, which are capable of performing *activities*. An activity $a \in \mathcal{A}ct$ is a pair (a, r), where $a \in \mathcal{A}$ is its action type, and $r \in \mathbb{R}_{\geq 0} \cup \{\top\}$ is the rate of the activity. This rate parameterises an exponential distribution that gives the duration of the activity, and if unspecified (denoted $\top$), the activity is said to be *passive*. This requires another component in cooperation to actively drive the activity. Because every activity has a duration, there is no non-determinism in PEPA, unlike in some other stochastic process algebras such as IMC [87].

PEPA terms have the following syntax:

$$\begin{aligned} C_S &:= (a,r).C_S \mid C_S + C_S \mid A \\ C_M &:= C_S \mid C_M \underset{L}{\bowtie} C_M \mid C_M / L \end{aligned}$$

We call a term C_S a *sequential component*, and a term C_M a *model component*. To define a PEPA model, we need to identify a particular model component that describes its initial configuration, which we call the *system equation*. The meaning of each combinator is as follows:

$$
\begin{array}{lcl}
Server & \overset{def}{=} & (request, \top).Server' \\
Server' & \overset{def}{=} & (serve, r_s).Server'' \\
Server'' & \overset{def}{=} & (response, r_d).Server \\
 & & \\
Client & \overset{def}{=} & (request, r_a).Client' \\
Client' & \overset{def}{=} & (response, \top).Client \\
 & & \\
\multicolumn{3}{l}{Server \underset{\{request, response\}}{\bowtie} (Client \parallel Client)}
\end{array}
$$

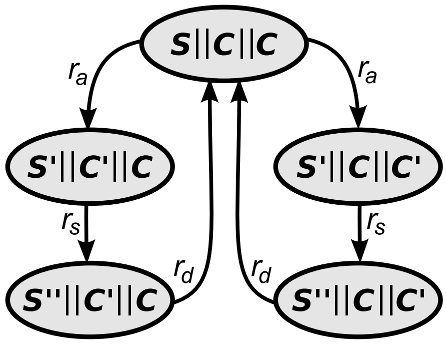


Figure 2.5: An example PEPA model and its underlying CTMC

- *Prefix* $((a,r).C)$: the component can carry out an activity of type a at rate r to become the component C.

- *Choice* $(C_1 + C_2)$: the system may behave either as component C_1 or C_2. The current activities of both components are enabled, and the first activity to complete determines which component proceeds. The other component is discarded.

- *Cooperation* $(C_1 \underset{L}{\bowtie} C_2)$: the components C_1 and C_2 synchronise over the cooperation set L. For activities whose type is not in L, the two components proceed independently. Otherwise, they must perform the activity together, at the rate of the slowest component. When $L = \{\ \}$, we write $C_1 \parallel C_2$.

- *Hiding* (C/L): the component behaves as C, except that activities with an action type in L are hidden, and appear externally as the unknown type τ.

- *Constant* $(A \overset{def}{=} C)$: the name A is assigned to the component C.

If a PEPA component can perform more than one activity of the same action type a, then to an observer it appears to be capable of performing a at the sum of the rates of these activities. We therefore need a notion of *apparent rate*. The apparent rate $r_a(C)$ of action type a in component P is defined as follows:

$$
\begin{array}{lcl}
r_a((b,r).C) & = & \begin{cases} r & \text{if } b = a \\ 0 & \text{if } b \neq a \end{cases} \\
r_a(C_1 + C_2) & = & r_a(C_1) + r_a(C_2) \\
r_a(C/L) & = & \begin{cases} 0 & \text{if } a \in L \\ r_a(C) & \text{if } a \notin L \end{cases} \\
r_a(C_1 \underset{L}{\bowtie} C_2) & = & \begin{cases} \min\{r_a(C_1), r_a(C_2)\} & \text{if } a \in L \\ r_a(C_1) + r_a(C_2) & \text{if } a \notin L \end{cases}
\end{array}
$$

$$\textsc{Prefix}\ \frac{}{(a,r).C \xrightarrow{(a,r)} C} \qquad\qquad \textsc{Const}\ \frac{C \xrightarrow{(a,r)} C'}{A \xrightarrow{(a,r)} C'}\ \ (A \overset{\mathit{def}}{=} C)$$

$$\textsc{Choice}_1\ \frac{C_1 \xrightarrow{(a,r)} C_1'}{C_1 + C_2 \xrightarrow{(a,r)} C_1'} \qquad\qquad \textsc{Choice}_2\ \frac{C_2 \xrightarrow{(a,r)} C_2'}{C_1 + C_2 \xrightarrow{(a,r)} C_2'}$$

$$\textsc{Sync}_1\ \frac{C_1 \xrightarrow{(a,r)} C_1'}{C_1 \underset{L}{\bowtie} C_2 \xrightarrow{(a,r)} C_1' \underset{L}{\bowtie} C_2}\ \ (a \notin L) \qquad \textsc{Sync}_2\ \frac{C_2 \xrightarrow{(a,r)} C_2'}{C_1 \underset{L}{\bowtie} C_2 \xrightarrow{(a,r)} C_1 \underset{L}{\bowtie} C_2'}\ \ (a \notin L)$$

$$\textsc{Sync}_3\ \frac{C_1 \xrightarrow{(a,r_1)} C_1' \qquad C_2 \xrightarrow{(a,r_2)} C_2'}{C_1 \underset{L}{\bowtie} C_2 \xrightarrow{(a,R)} C_1' \underset{L}{\bowtie} C_2'}\ \ (a \in L),\ \text{ where } R = \frac{r_1 r_2 \min\{r_a(C_1), r_a(C_2)\}}{r_a(C_1) r_a(C_2)}$$

$$\textsc{Hide}_1\ \frac{C \xrightarrow{(a,r)} C'}{C/L \xrightarrow{(a,r)} C'/L}\ \ (a \notin L) \qquad\qquad \textsc{Hide}_2\ \frac{C \xrightarrow{(a,r)} C'}{C/L \xrightarrow{(\tau,r)} C'/L}\ \ (a \in L)$$

Figure 2.6: The operational semantics of PEPA

The reason for summing the apparent rates in the case of a choice is that both activities take place in parallel — the first one to complete determines which choice is made. In other words, we are taking the minimum of two exponential distributions with rate parameters r_1 and r_2, which is itself an exponential distribution with rate parameter $r_1 + r_2$. This is in contrast to the cooperation combinator, where we take the minimum of the *rates*, rather than the distributions. The justification for this interpretation of synchronisation is discussed at length in [90].

The operational semantics of PEPA is shown in Figure 2.6, and defines a labelled multi-transition system for a PEPA model. If a component C can evolve to component C' by a series of transitions — namely, $C \xrightarrow{(a_1,r_1)} \cdots \xrightarrow{(a_n,r_n)} C'$ — then we call C' a *derivative* of C. The *derivative set* $\mathrm{ds}(C)$ is the set of all derivatives of C. These derivatives, and the transitions between them, form a *derivation graph*. Since the duration of a transition in this graph is exponentially distributed, we can sum the rates on all transitions between two states in the derivation graph to generate a CTMC.

An example of a PEPA model, along with its underlying CTMC, is shown in Figure 2.5. It consists of three sequential components — a server and two clients. Each client synchronises with the server over the *request* and *response* action types, but is independent of the other. This means that the server makes an internal choice between which client to serve, as is seen in the CTMC. The rates are parameterised as r_a, r_s, and r_d, and if we substitute them for real values (in $\mathbb{R}_{\geq 0}$), we get a concrete CTMC.

As an alternative to the Markovian semantics of PEPA, a semantics has been given in terms of ordinary differential equations [92]. When we have several copies of the same sequential component in the system equation, such as the clients in the example, the CTMC records precisely which component is in each state. The differential equations, however, record just the *number* of components in each state[2]. Since differential equations are continuous, this represents a fluid-flow approximation, capturing the evolution over time of the average number of components in each sequential state. PEPA models can also be analysed using stochastic simulation [33].

2.5.3 The State Space Explosion Problem

The problem with compositional formalisms such as PEPA is that they allow us to very easily define Markov chains that are exponentially large with respect to the size of their description. For example, if a model has six components, each having ten states, then the underlying Markov chain can have up to a million states. Since many models of real-world systems have extremely large state spaces, there has been a huge amount of research looking at ways to reduce the size of models. The best available tools, such as the PRISM model checker [114] can handle models of size 10^7–10^8 states, and can even handle up to 10^{11} states under special circumstances (i.e. if the model exhibits a certain structure). However, many models can be much larger that this in practice, and we need a way of dealing with them.

There are two key ideas to reducing the size of a Markovian model — *decomposition* and *aggregation*. Decomposition structurally breaks apart the model into components that can be solved separately. Aggregation, on the other hand, combines states that exhibit similar behaviour, to create a smaller model that is easier to solve.

The best example of a decomposition technique is when a Markov chain has a *product form solution*. This means that it is possible to decompose the Markov chain into components, so that its steady state solution can be expressed as a product of

[2] Note that this is not just a counting abstraction of a Markov chain, where we aggregate states the correspond to the same number of each type of component, since they are lumpable. Instead of having a discrete state stochastic model (given by a CTMC), we have a continuous state deterministic model (given by a system of differential equations).

the components' solutions. Product forms have been widely studied in the context of queueing networks [24], and also for SAN [70] and PEPA [95, 45]. The difficulty is that they are rare in practice, requiring particular forms of interaction between components.

Aggregation techniques, on the other hand, are concerned with grouping states into partitions, which can then be represented by a single state in a reduced model. In general, it is not possible to combine states and still end up with a Markov chain, unless certain strict conditions apply. A partitioning of a Markov chain is *lumpable* [108] if the probability of moving between partitions is independent of the particular state we are in. In this case, the abstraction is said to be *exact* — if we solve the lumped Markov chain, the steady state probability of being in one partition is equal to the sum of the steady state probabilities in the original Markov chain of the states in that partition.

There are several different variants of lumpability, including ordinary lumpability [108], exact lumpability [158], strict lumpability [158] and weak lumpability [108], which we will discuss in more detail in Chapter 5. If the Markov chain is not quite lumpable, but the probability of moving between partitions is *almost* the same for every state in a partition, it is said to be *quasi-lumpable* [57]. The aggregated Markov chain will approximate the aggregated solution of the original.

An alternative basis for aggregation is *insensitivity* [154, 155]. If we aggregate a sequence of states in a Markov chain, then the sojourn time of the state will satisfy a more general *phase type distribution*, rather than being exponential. Although this no longer satisfies the Markov property, we can safely replace this distribution with an exponential distribution of the same mean, only if we can be certain that the activity will never be interrupted — there can be no other components that can independently perform activities at the same time as the aggregated activity. This ensures that the steady state probability distribution is the same, but not the transient probabilities. More precisely, the steady state distribution of the Markov chain is insensitive to the actual distribution of the activity, and depends only on its mean duration.

Rather than performing a purely structural decomposition, or an abstraction, there are also a number of methods for solving the steady state distribution of a Markov chain, based on both. These Decomposition/Aggregation (D/A) techniques [159, 126] work by structurally decomposing the Markov chain into distinct partitions. Each partition can be analysed separately, but the Markov chain can also be aggregated, so that each partition is represented by a single state. By taking the product of the local and aggregate steady state probabilities, we can approximate the steady state distribution of the original Markov chain.

Time scale decomposition [11, 49, 163] is one application of this technique, based on separating transitions into those with fast rates and those with significantly slower rates. By ignoring the slow transitions, we can partition the state space into connected components on the basis of the fast transitions. This leads to a nearly completely decomposable [57] transition matrix, which can be solved by the method of Decomposition/Aggregation.

Another technique is response time analysis [100, 40], where we look for a single-in single-out (SISO) cut through the transition system. This allows us to consider the system as two parts, and we can solve each part independently by aggregating all the states in the other to a single state. Feeding the steady state information from each part back into the other, we can iteratively solve each part in turn, until we (hopefully) converge on the steady state distribution of the entire system.

2.6 Performance Analysis from Specifications

In relation to performance engineering, there has been a great deal of interest in extracting performance models from specification formalisms. The idea is to make use of existing design documents — with which software developers are familiar — rather than requiring them to use separate modelling languages. Since UML [152] is the specification language most widely used in development, much of this research has focused on extracting various types of performance model from UML diagrams. UML provides a wide range of diagrams, allowing both structural and behavioural properties to be specified, both from the high level perspective of the user, and a lower level implementation perspective. For example, *use case diagrams* describe a user-level view of a particular interaction with the system, whereas *class diagrams* present a lower-level structural view of the system.

Although these diagrams are inherently functional — they describe *what* the system should do rather than *how fast* it should do it — performance information such as rates can be added as annotations [143, 184]. The Object Management Group has to date produced two specifications for UML performance annotations — in 2005 it released the UML Profile for Schedulability, Performance and Time Specification (SPT) [81], which was superceded in 2008 by the UML Profile for Modelling and Analysis of Real-Time and Embedded Systems (MARTE) [82]. The motivation behind these schemes is to take advantage of existing UML diagrams, so that designers do not need to learn to use additional performance-specific diagrams.

Each type of UML diagram can support performance engineering in different ways. For example, while use case diagrams are too high level to usefully describe performance properties of the system, they can be used to identify workloads — the input to a performance model [144]. It was also suggested in [144] that *sequence diagrams* are useful for describing transient behaviour, whereas *collaboration diagrams* are of more use in describing a system's longer term steady state behaviour.

The SPT performance profile allows both instances of objects and the actions they perform to be labelled with performance information. This is specified by a *stereotype*, which describes the general performance characteristics of the object/action, along with a number of parameters that define its specific behaviour. Performance analysis is carried out with respect to a *scenario* — an ordered sequence of operations that the system performs — and a *workload* — the environment in which the scenario takes place. The MARTE performance profile improves upon SPT by updgrading to UML2, including real-time mechanisms and richer models of time, and supporting different analysis domains, such as energy consumption and reliability.

To make it easier to transform annotated UML diagrams into performance models, and to provide a common framework for the different modelling formalisms, the Core Scenario Model has been proposed as an intermediate language [138]. In addition, UML diagrams have been translated into a wide range of performance modelling formalisms, including queueing networks [144, 47], stochastic Petri nets [118], PEPA [41, 179], and MoDeST [88]. An alternative approach is taken by UML-Ψ [122], which uses discrete event simulation to evaluate the performance characteristics of annotated use case, activity and deployment diagrams. There is a vast amount of literature on performance modelling and UML, and this is just a small sample.

In the context of PEPA, it was shown in [41] how to generate a model from statechart diagrams (for sequential components) and a collaboration diagram (for the system equation). By automatically translating performance-annotated UML to PEPA, and reflecting the analysis results back to the UML model, the designer does not require an in-depth knowledge of performance modelling. More recently, PEPA models have also been generated from MARTE-annotated sequence diagrams [179].

2.7 Performance Analysis from Source Code

Unlike in the qualitative world, there has been very little work in relating performance analysis to program code. This is largely because performance properties are usually much more global — depending on the context in which a program is run, as much as the program code itself. In general, it is difficult to apply the same sort of abstraction techniques that led to the success of Microsoft's static driver verifier, since control flow decisions depend on the actual values of variables. Since we are concerned with the *probability* of each control flow decision, rather than the *possibility*, we would need information about the distribution of values of the program's variables, at the time that each control flow decision is made.

The work in [65] avoids this problem by completely abstracting away from a program's data flow. Instead, they abstract control flow to a 'service effect automaton', using annotations to describe the probabilities of control flow decisions. Calls to other functions or services have a duration, given as a random variable, and the duration of the entire automaton is calculated by composing these. Using the Quality of Service Modelling Language (QML) [74], properties of the required execution-time distribution can be specified and then verified.

In contrast to program code, there has been considerably more work that focuses on higher level languages — for example those used for *orchestrating* distributed systems such as web services. One such language is the Web Services Business Process Execution Language (WS-BPEL) [135], which is used to describe how web services interact with external systems. WS-BPEL specifications have been mapped to a number of performance modelling formalisms, including layered queueing networks [54] and PEPA [183]. A similar notion to orchestration is found in parallel programming systems, where algorithmic skeletons are often used to describe the structure of parallel and sequential operations. There has been some work on translating algorithmic skeletons to performance models — for example, to PEPA [185].

The problem with these approaches is that all the required performance information must be specified in annotations. This is reasonable at the specification level, where we incorporate performance criteria into the design, but once we have an implementation we should be able to infer at least some information from the code itself. One of the goals of this thesis is to illustrate an approach for doing this, and we will describe a technique for generating PEPA models from program code in Chapter 4. Before doing so, however, we need to introduce the language on which we will base our analysis — the Simple Imperative Remote Invocation Language (SIRIL).

Chapter 3

A Language for Distributed Programs with Remote Invocation

If there is a single message that we want to present in this thesis, it is that developers should be able to reason about performance during the development process, without requiring years of expertise in mathematical modelling. For our goal of *performance-driven* development to succeed, we need to build tool support for relating program code to performance models. This means — as we motivated in the previous chapter — being able to automatically derive such a model from the code. The challenge of doing this for real languages is immense, and so for the purposes of this thesis we will introduce and work with a simple programming language, whose programs we can formally analyse in order to build probabilistic models of their behaviour.

Let us take a moment to consider why we should want a *probabilistic* model of a program — after all, is software not inherently designed to be deterministic? The hardware that it runs on is deterministic (if we ignore the effect of cosmic rays and extra-terrestrials), and users expect it to behave in a predictable, deterministic manner. The answer ultimately boils down to *abstraction* — the software running on an average computer is so complex that any attempt at reasoning about its precise behaviour is futile. For example, to predict the running time of even the simplest program, we would need to know the behaviour of the operating system, the hardware, and the exact state of the system when we execute it. If we take empirical measurements of the running time, however, we will find that it follows some *distribution*. That is to say, we will observe the underlying deterministic system as a probabilistic system.

In general, this sort of probabilistic behaviour is a result of uncertainty in the environment in which the program runs. This might be uncertainty about the state of the operating system (in particular, the process scheduler), but it could just as easily be the state of the network that eludes us, leading to uncertainty in the transit time of a packet. Yet even in this probabilistic world we can have difficulty reasoning about the behaviour we observe. The probability distribution of the environment will, in general, be very complex (and likely unrepresentable), and the distribution of the program's output even more so. Because of this, there is a need for a second layer of abstraction, where we approximate these probability distributions. For example, we may only be interested in the mean and variance of a particular variable, in which case we can abstract away all the higher order moments. Abstraction is the key to building a model of the system that we can actually work with — the challenge is to make it computationally simple enough without losing the information we care about.

Before we look at this latter form of abstraction — which is the subject of Chapter 4 — we need to establish the concrete language that we will work with. In this chapter, we introduce an imperative programming language called SIRIL, which has integer variables, and remote invocation features in addition to imperative constructs. In Section 3.1 we present the syntax of SIRIL, and give an example of a simple client-server system. We present some background to probabilistic semantics of programming languages in Section 3.2, before describing the semantics of SIRIL in Section 3.3. Finally, we give a probabilistic interpretation of this semantics in Section 3.4, and discuss how we might obtain a compositional performance model in Section 3.5.

3.1 The Language SIRIL

Before we can hope to analyse real-world languages such as Java and C, we need to build techniques that work for a much simpler language. In particular, we need a language that is as small as possible, whilst still containing the features that we want to analyse. Since the aim of this thesis is to analyse the performance characteristics of distributed software, the most important features to include are *concurrency* and *remote invocation*. We will not consider pointers and aliasing, recursion, or looping behaviour. This latter restriction is a somewhat severe consequence of our analysis, and we hope to be able to relax it in the future. We will show how to extend SIRIL with loops in Appendix A, and discuss the difficulties that arise when we apply our analysis in this setting.

The Simple Imperative Remote Invocation Language (SIRIL) is at its core an imperative language with integer variables and conditional branching. A SIRIL program consists of a number of statically defined and immutable *objects*, each providing an *interface* in terms of a set of method definitions. Methods on different objects can be invoked in the style of a *remote procedure call* (RPC). As an example, we might have a *Server* object that provides a method *requestPage*$(\cdot)$, and this can be remotely invoked using the syntax *Server.requestPage*(x) for some local variable x. To state this more formally, a SIRIL program consists of the following elements:

1. A finite set of immutable *objects* (identified by strings), which we will denote by the metavariable O. Objects exist for the lifetime of the SIRIL program, and cannot be created or destroyed.

2. A finite set of *methods* associated with each object, which we will denote by $O.f$. Each method has a fixed *arity*, specified at the time of definition, which takes the following form (we will shortly define the syntax for commands C):

$$O.f(X,\ldots,X)\,\{C\}$$

Since each method is defined statically, we will write $\mathit{def}(O.f)$ to denote the definition (in the above form) of the method $O.f$.

3. A finite set of *variables* which we will denote by X. We assume that there are no overlaps between variable names in different methods (including those corresponding to arguments), which means that we have a single set of variables for the entire system. Since there are no shared variables — the objects do not maintain state — this can be achieved by alpha-renaming. For reasons that will be made clear in Section 3.3, each method also has an implicit variable, $X_{O.f}$, in which to store the value it returns.

The objects in a SIRIL program can be thought of as existing in parallel, but nothing happens until one of their methods is invoked. An object represents a single *resource*, which means that only one of its methods can be invoked at any one time. An *instantiation* of a SIRIL program is a call to one of its methods, $O.f(x_1,\ldots,x_n)$, where $x_1,\ldots,x_n$ are the initial arguments, or inputs. For the purposes of this chapter, we will just consider a single instantiation, but when we come to generate performance models in Chapter 4, it will be useful to also consider both repeated sequential instantiations — such as a client repeatedly requesting data from a server — and parallel instantiations — such as several clients attempting to connect to a server at once.

In SIRIL, arithmetic expressions have the following syntax ($c \in \mathbb{Z}$ are constants):

$$E \quad ::= \quad c \mid X \mid -E \mid E+E \mid c \times E$$

It is important to note that we only allow linear expressions — we cannot multiply two variables together in an expression. This is not a necessary condition for the semantics in this chapter, but is required for our analysis in the next chapter. Our motivation is to focus on control-flow branching and communication between objects, rather than more complex arithmetic. However, since non-linear arithmetic can be encoded in loops, the omission of these two features is related.

The Boolean expressions in the language are as follows:

$$B \quad ::= \quad \mathtt{true} \mid X < c \mid X \leq c \mid \neg B$$

Without loss of generality, we only allow a variable to be compared with a constant — we can always construct more complex comparisons by first defining a new variable. Furthermore, conjunction and disjunction of conditions can be expressed by the nesting of `if`-statements, so are not necessary as primitives.

Commands have the following form, where q is a real number in the interval $[0,1]$:

$$\begin{array}{rcl} C & ::= & \mathtt{skip} \mid \mathtt{return}\ E \mid X \mathrel{\mathtt{:=}} E \mid X \mathrel{\mathtt{:=}} f(X,\ldots,X) \\ & \mid & C\ \mathtt{;}\ C \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \\ & \mid & \mathtt{pif}\ q\ \mathtt{then}\ C\ \mathtt{else}\ C \mid X \mathrel{\mathtt{:=}} O.f(X,\ldots,X) \end{array}$$

Most of the above is self-explanatory, but branches and method calls require some further explanation. We include two types of conditional statement — a classical version, `if`, where the branch is determined by a Boolean condition B, and a probabilistic version, `pif`, where the first branch is taken with probability q, and the `else`-branch with probability $1-q$. Note that probabilistic branching can be viewed as a special case of conditional branching, since we could encode it using an external random variable (in essence, a random number generator). It is useful to have this as a primitive in our language, however, since it allows us to directly deal with more abstract programs — for example, in the case of partially written code, where we might use a probabilistic branch as a placeholder for a conditional branch.

Method calls are either *local* ($X \mathrel{\mathtt{:=}} f(X,\ldots,X)$) or *remote* ($X \mathrel{\mathtt{:=}} O.f(X,\ldots,X)$), where the former is shorthand for invoking a different method on the same object as that of the current method. The fact that we use *synchronous* communication, by way of remote procedure calls, may seem counter to the reality that systems built on top of a packet-switched network layer (such as IP) are inherently *asynchronous*. There are two main motivations for this choice. The first is that many distributed systems —

such as web services — are built on top of middleware that provides the abstraction of synchronous communication. In other words, this is a natural programming paradigm for a large number of applications. The second motivation is that asynchronous communication could be encoded in our language if we were to allow both stateful objects and looping behaviour — the latter is needed to implement polling. This is not possible in our language at present, for reasons that we will touch upon in the conclusions of this thesis, but it provides an interesting direction for future research — in particular, see Appendix A for a discussion on adding loops to the Siril language.

It is important to note that the syntax of Siril does not in itself forbid recursion, even though we wish to impose such a restriction so that we can feasibly carry out a performance analysis of Siril programs in the next chapter. To formally prohibit recursive programs — including those using multi-step recursion — we first need to determine which methods can call one another. We can over-approximate the set of methods that a given method calls as follows — $\mathit{Call}(\mathit{def}(O.f))$ returns the set of method names that occur syntactically in the method $O.f$:

$$
\begin{array}{lcl}
\mathit{Call}(O.f(X_1,\ldots,X_n)\{\,C\,\}) & = & \mathit{Call}_{O.f}(C) \\[1em]
\mathit{Call}_{O.f}(\mathtt{skip}) & = & \{\,\} \\
\mathit{Call}_{O.f}(\mathtt{return}\,E) & = & \{\,\} \\
\mathit{Call}_{O.f}(X := E) & = & \{\,\} \\
\mathit{Call}_{O.f}(X := f'(X_1,\ldots,X_n)) & = & \{\,O.f'\,\} \\
\mathit{Call}_{O.f}(X := O'.f'(X_1,\ldots,X_n)) & = & \{\,O'.f'\,\} \\
\mathit{Call}_{O.f}(C_1\,;\,C_2) & = & \mathit{Call}_{O.f}(C_1) \cup \mathit{Call}_{O.f}(C_2) \\
\mathit{Call}_{O.f}(\mathtt{if}\;B\;\mathtt{then}\;C_1\;\mathtt{else}\;C_2) & = & \mathit{Call}_{O.f}(C_1) \cup \mathit{Call}_{O.f}(C_2) \\
\mathit{Call}_{O.f}(\mathtt{pif}\;q\;\mathtt{then}\;C_1\;\mathtt{else}\;C_2) & = & \mathit{Call}_{O.f}(C_1) \cup \mathit{Call}_{O.f}(C_2)
\end{array}
$$

This allows us to define a call graph for a Siril program, as follows:

Definition 3.1.1. *The* syntactic call graph *of a* Siril *program is a pair* (V,E)*, where* $V = \{\,O_i.f \mid 1 \leq i \leq M, f \in \mathcal{F}(O_i)\,\}$ *is a set of nodes corresponding to each method* $O.f$ *in the program, and* $E \subseteq V \times V$ *is an edge relation: there is an edge* $(O_1.f_1, O_2.f_2) \in E$ *iff* $O_2.f_2 \in \mathit{Call}(\mathit{def}(O_1.f_1))$*.*

Using this, we can state a well-formedness condition for Siril programs, which we call *syntactically non-recursive*. Crucially, no method in a syntactically non-recursive Siril program will ever recursively call itself, after any number of steps. It is possible, however, to write a Siril program that does not exhibit recursion, but is not syntactically non-recursive.

Definition 3.1.2. *A* Siril *program is* syntactically non-recursive *if its syntactic call graph is acyclic.*

In the remainder of this thesis, we will implicitly assume that we are only considering syntactically non-recursive Siril programs. This is important to remember when we present the denotational semantics for the sequential fragment of Siril in Section 3.2, as it would otherwise be ill-founded.

3.1.1 Writing Programs in Siril

In order to construct some readable examples of Siril programs, we will introduce a few derived terms that extend the syntax we presented. These are simply shorthand notations, and do not increase the expressive power of the language. For arithmetic and Boolean expressions, we can define the following:

$$\begin{array}{rcl@{\qquad}rcl} E - E & \triangleq & E + (-E) & \mathtt{false} & \triangleq & \neg\mathtt{true} \\ X \geq c & \triangleq & \neg(X < c) & X > c & \triangleq & \neg(X \leq c) \end{array}$$

In addition, we will often want to write conditions that test whether a variable is equal to a constant, or that compare two arbitrary arithmetic expressions (as opposed to just a variable and a constant). These can be encoded in Siril as follows:

$$\begin{array}{rcl} \mathtt{if}\,(X = c)\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2 & \triangleq & \mathtt{if}\,(X \leq c)\,\mathtt{then} \\ & & \quad(\mathtt{if}\,(X \geq c)\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2) \\ & & \mathtt{else} \\ & & \quad C_2 \\ \\ \mathtt{if}\,(E_1 \trianglelefteq E_2)\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2 & \triangleq & Y := E_1 - E_2; \\ & & \mathtt{if}\,(Y \trianglelefteq 0)\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2 \end{array}$$

where $\trianglelefteq \in \{<, \leq, >, \geq, =\}$, and Y is a fresh variable name. Note that in the latter case, we need to introduce a temporary variable on which we perform the actual comparison. We will also avoid writing the `else` clause of a conditional, when the body is `skip`:

$$\begin{array}{rcl} \mathtt{if}\,B\,\mathtt{then}\,C & \triangleq & \mathtt{if}\,B\,\mathtt{then}\,C\,\mathtt{else}\,\mathtt{skip} \\ \mathtt{pif}\,q\,\mathtt{then}\,C & \triangleq & \mathtt{if}\,B\,\mathtt{then}\,C\,\mathtt{else}\,\mathtt{skip} \end{array}$$

As an example of a Siril program (making use of the shorthands that we have introduced), consider Figure 3.1. This consists of two objects, `Client` and `Server`, and is instantiated by a call to `Client.buy`. The client first queries the server to determine

```
Client.buy(quantity, cash) {
  price := Server.getPrice(quantity);
  if (price ≤ cash) then
    success := Server.buy(quantity);
    if (success = 1) then
      cash := cash − price;
  return cash
}

Server.getPrice(quantity) {
  if (quantity > 0) then
    if (quantity > 10) then
      return 8 × quantity
    else
      return 10 × quantity
  else
    return 0
}

Server.buy(quantity) {
  max_order := 100;
  if (quantity ≤ max_order) then
    return 1
  else
    return 0
}
```

Figure 3.1: An example of a client-server system in Siril

```
LossyServer.request(data) {
  response := -1;
  pif (0.9) then
    response = Server.request(data);
  return response
}

LoadBalancer.request(req) {
  pif (0.5) then
    response = ServerA.request(data)
  else
    response = ServerB.request(data);
  return response
}
```

Figure 3.2: Two examples of network behaviour, corresponding to packet loss and load balancing

the cost of purchasing a number of items, specified by the `quantity` argument. If there are sufficient funds available, it then attempts to buy that number of items, although the server will reject this if it is greater than the maximum allowed order size. If the transaction is successful, `Client.buy` computes and returns the new available funds, and otherwise returns the original value of `cash`.

To perform a useful analysis of the overall system, it is essential to know something about the network behaviour. For example, the client-server system only becomes interesting when we consider factors such as the time taken to send a message on the network, and/or the probability of packet loss. There are two ways in which we might include such information — build it into the semantics of the language, or encode it in the program itself. Since there is no way to directly talk about time in Siril — which is really an externally controlled factor — we will take the semantic approach in Section 3.4. For network behaviours that are time-independent, such as when there is a fixed probability of packet loss, we will assume that they are directly encoded in the program — this allows more flexibility.

Two examples of such encodings are shown in Figure 3.2, where we build wrappers around a `Server` object. The first, `LossyServer`, implements a network where there is a 10% chance of a packet being lost. The second is a load-balancer, which forwards any request it receives to either `ServerA` or `ServerB` with equal probability.

3.2 Probabilistic Semantics of Programs

There are several ways in which we can assign a meaning to a program. The two most widely used approaches are *denotational semantics*, where the program is viewed as a mathematical function between an input and an output domain, and *operational semantics*, where the program is viewed as a transition system. The difference between these approaches is apparent when we look at the treatment of loops. In a denotational semantics, a loop is the least fixed point of the function that describes a single iteration of its body. In an operational semantics, a loop is the transitive closure of the transition relation describing its body.

Since Siril does not contain loops, we could choose to describe its semantics either operationally or denotationally. In this section, we will present a *denotational* semantics for single-threaded Siril programs, because it allows us to describe the operation of a sequence of commands as a single mathematical operator. This is advantageous with respect to our ultimate goal of constructing a performance model from a Siril program, since it avoids describing each command as a separate execution step, which would correspond to a separate state in the performance model. In Section 3.3, we will extend this to multi-threaded Siril programs, by adding an automaton structure to the denotational semantics, which will allow us to separate the execution steps of different threads. We could have achieved a similar result using a big-step operational semantics (otherwise known as a natural semantics) [103], but this would have made our treatment of multi-threading more cumbersome.

In a classical — non-probabilistic — denotational semantics, partially ordered domains are used to describe the input and output of a program, as per Scott and Strachey [173]. The meaning of a program is then a continuous function between these domains. This approach was first extended to a probabilistic setting by Kozen [112], who replaced the partially ordered domains with vector spaces.

Kozen presents two alternative approaches to developing such a semantics. The first approach is to treat the program's variables as *random variables*. The output is then a new random variable that is a function of the inputs — to sample the output, we first sample the inputs, and then apply the program deterministically. This has several shortcomings, most notably when the program itself contains a source of randomness, in which case we must keep track of a stack of random variables. The second approach is for the input to be a measure on a measurable space. In this setting, a program is no longer a map between an input and an output *value*, but a map between an input and an output *probability measure*. These measures form a partially ordered vector space, and we can directly extend the Scott-Strachey style of denotational semantics to this setting.

For the purpose of this thesis, we will only look at this second approach of Kozen. To use this to describe a probabilistic semantics of Siril, we first need to introduce some definitions from measure theory [42]:

- A *σ-algebra* on a set X is a subset of $\mathcal{P}(X)$ that contains $\emptyset$ and X, and is closed under countable union and countable intersection.
- A *measurable space* (X,σ_X) is a set X with a σ-algebra σ_X. The elements of σ_X are called the *measurable subsets* of X.
- A *measurable function* is a function $f\colon (X,\sigma_X) \to (Y,\sigma_Y)$ that is well behaved; namely, $f^{-1}(\sigma_Y) \subseteq \sigma_X$.
- A *measure* is a countably additive function $\mu\colon \sigma_X \to \mathbb{R}_{\geq 0}$ over a measurable space (X,σ_X). We can perform addition and and scalar multiplication of measures on the same measurable space in a pointwise fashion: for measures over (X,σ_X), $(\mu_1+\mu_2)(x) = \mu_1(x)+\mu_2(x)$ and $(c\mu)(x) = c(\mu(x))$, for all $x \in \sigma_X$ and $c \in \mathbb{R}_{\geq 0}$.
- The *total weight* of a measure μ on a measurable space (X,σ_X) is given by $\mu(X)$.
- A *probability measure* is a positive measure with total weight 1.
- A *measure space* (X,σ_X,μ) is a measurable space (X,σ_X) equipped with a measure μ. We write $\boldsymbol{B}(X,\sigma_X)$ for the set of all measures on the measurable space (X,σ_X) — this defines a vector space, due to the addition and scalar multiplication operators over measures.
- A *probability space* is a measurable space equipped with a probability measure.
- A *random variable* is a measurable function whose domain is a probability space.

- The *Lebesgue measure* $m^*(X)$ of a set $X \subseteq \mathbb{R}$ is given as follows, where $l(I) = b - a$ for any interval $I = [a,b]$, $(a,b]$, $[a,b)$ or (a,b):

$$m^*(X) = \inf \left\{ \sum_{n=1}^{\infty} l(I_n) \;\middle|\; I_n \text{ is an interval} \wedge X \subseteq \bigcup_{n=1}^{\infty} I_n \right\}$$

- A set $X \subseteq \mathbb{R}$ is *Lebesgue-measurable* if for every set $Y \subseteq \mathbb{R}$:

$$m^*(Y) = m^*(Y \cap X) + m^*(Y \cap (\mathbb{R} \backslash X))$$

  We write $\mathcal{M}$ for the set of all Lebesgue-measurable[1] subsets of $\mathbb{R}$.

- A *Banach space* $(V, ||\cdot||)$ is a complete normed vector space — it consists of a vector space V, equipped with a norm $||\cdot|| : V \to \mathbb{R}_{\geq 0}$ on elements $v \in V$ such that for every sequence $v_1, v_2, \ldots$ in V, such that $\lim_{i\to\infty} ||v_{i+1} - v_i|| = 0$ (i.e. it is a Cauchy sequence), has a limit in V.

Consider the vector space $\boldsymbol{B}(X, \sigma_X)$ of measures over the measurable space (X, σ_X). If we define a norm $||\cdot||$ on the measures $\mu \in \boldsymbol{B}(X, \sigma_X)$ such that $||\mu|| = \mu(X)$ (i.e. the norm of a measure is its total weight), then $(\boldsymbol{B}(X, \sigma_X), ||\cdot||)$ forms a Banach space [112]. The completeness of the norm follows from the completeness of the non-negative reals $\mathbb{R}_{\geq 0}$, and the countable additivity of measures.

We can now define a probabilistic denotational semantics $[\![\cdot]\!]_p$, for single-threaded Siril methods, in the style of Kozen [112] — we will extend this to include remote procedure calls in Section 3.3. It is important to recall that the syntax of Siril allows only allow integer-valued variables — the semantics we present, however, will be in terms of *continuous measures*. This means that our semantics will lift the domain of the variables from the integers to the reals, and therefore is more general with respect to the the measures over the state of the variables that are possible.

A method in Siril operates on a fixed set of N variables, including its arguments, and each of these is considered to be real-valued for the purposes of the semantics. Let $\boldsymbol{B} = \boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$ be the set of measures on the Cartesian product of N copies of the measurable space $(\mathbb{R}, \mathcal{M})$ — $\mathcal{M}^{(N)}$ is the σ-algebra generated from the set of products of N sets from $\mathcal{M}$ (the Lebesgue-measurable subsets of $\mathbb{R}$). Then we will

[1] To construct an example of a non Lebesgue-measurable subset of $\mathbb{R}$, define an equivalence relation on $R \subset [0,1] \times [0,1]$ — we say that $(x,y) \in R$ iff $x - y$ is rational. We can then use R to partition $[0,1]$ into disjoint equivalence classes A_i (of which there are countably many), and use the axiom of choice to select a single characteristic element a_i from each A_i: the set of all such a_i is not Lebesgue-measurable. This is called a Vitali set. For a proof that Vitali sets are not Lebesgue-measurable, see the Appendix of [42].

$$\begin{array}{lcllcl}
[\![c]\!](\boldsymbol{x}) & = & c & [\![\mathtt{true}]\!](\boldsymbol{x}) & = & \mathrm{true} \\
[\![X_i]\!](\boldsymbol{x}) & = & \boldsymbol{x}_i & [\![X_i < c]\!](\boldsymbol{x}) & = & \boldsymbol{x}_i < c \\
[\![-E]\!](\boldsymbol{x}) & = & -[\![E]\!](\boldsymbol{x}) & [\![X_i \leq c]\!](\boldsymbol{x}) & = & \boldsymbol{x}_i \leq c \\
[\![E_1 + E_2]\!](\boldsymbol{x}) & = & [\![E_1]\!](\boldsymbol{x}) + [\![E_2]\!](\boldsymbol{x}) & [\![\neg C]\!](\boldsymbol{x}) & = & \neg[\![C]\!](\boldsymbol{x}) \\
[\![cE]\!](\boldsymbol{x}) & = & c[\![E]\!](\boldsymbol{x}) & & &
\end{array}$$

Figure 3.3: Deterministic semantics of arithmetic and Boolean expressions in Siril

define $[\![\cdot]\!]_p : (\boldsymbol{B}, \|\cdot\|) \to (\boldsymbol{B}, \|\cdot\|)$ as a *continuous linear operator* on the Banach space $(\boldsymbol{B}, \|\cdot\|)$ (we will prove that this is the case in Theorem 3.2.1). Note that the linearity of the semantics is unrelated to our restriction of Siril to linear arithmetic expressions, and would hold for more general expressions, such as in [112].

The probabilistic denotational semantics $[\![O.f \mid C]\!]_p$ of a Siril command C, within a method $O.f$, transforms a measure μ over the variables in the method into a new measure μ', which captures the effect of applying the command C. We will consider each type of command in turn.

Let us begin with assignment. The deterministic behaviour of such a command operates on a *state* of the method's variables — if there are N variables, $X_1, \ldots, X_N$, then we can represent this as a vector $\boldsymbol{x} \in \mathbb{R}^N$, where $\boldsymbol{x}_i$ records the value of the variable X_i. An assignment $X_i := E$ transforms this vector, by substituting the value of X_i with the value that E evaluates to. We can formally write this deterministic semantics $[\![\cdot]\!] : \mathbb{R}^N \to \mathbb{R}^N$ as follows:

$$[\![X_i := E]\!](\boldsymbol{x}) = \boldsymbol{x}\{\, [\![E]\!](\boldsymbol{x}) \,/\, \boldsymbol{x}_i \,\}$$

where the definition of $[\![E]\!]$ is shown in Figure 3.2. To construct a probabilistic, rather than a deterministic semantics, we need instead to map between two *measures* in $\boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$. Specifically, given a measure μ, the new measure on a set $X \in \mathcal{M}^{(N)}$ should be the original measure applied to the set of states that can lead to states in X following the assignment:

$$[\![O.f \mid X_i := E]\!]_p(\mu) = \mu \circ [\![X_i := E]\!]^{-1}$$

The semantics of a sequential composition of two commands is the composition of the operators on measures given by their semantics:

$$[\![O.f \mid C_1 \,;\, C_2]\!]_p = [\![O.f \mid C_2]\!]_p \circ [\![O.f \mid C_1]\!]_p$$

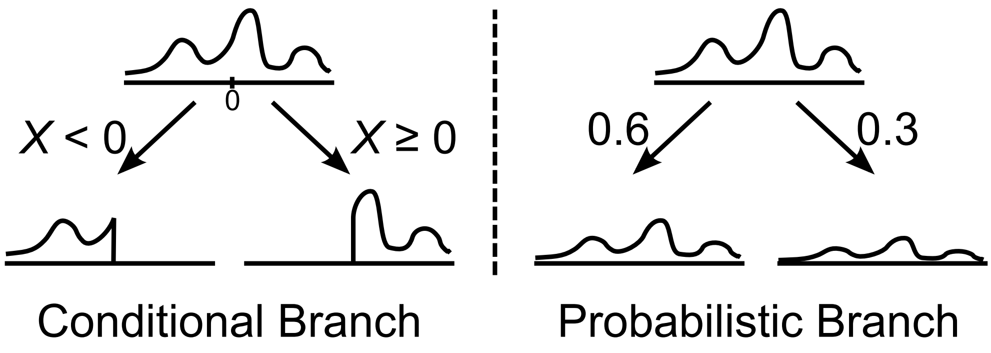


Figure 3.4: The effect of conditional and probabilistic branching on a measure

To describe the semantics of `if`-statements, we first need to describe the semantics of conditions. For a Boolean expression B, we will write $[\![B]\!]$ to denote the set of valuations that satisfy B — namely, $[\![B]\!] = \{\boldsymbol{x} \mid [\![B]\!](\boldsymbol{x}) = \text{true}\}$. Using the notation μ_Y to mean $\mu_Y(Z) = \mu(Y \cap Z)$, we define $e_{[\![B]\!]}$ to be a linear operator on measures, such that $e_{[\![B]\!]}(\mu) = \mu_{[\![B]\!]}$. The intuition behind this is to throw away all the unreachable environments — that do not satisfy B — before applying the measure μ. The semantics of an `if`-statement can now be defined as follows:

$$[\![O.f \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]_p = [\![O.f \mid C_1]\!]_p \circ e_{[\![B]\!]} + [\![O.f \mid C_2]\!]_p \circ e_{[\![\neg B]\!]}$$

This splits the probability mass into two parts, according to the condition, passing each part down the corresponding branch. It then recombines the two measures, when the control flow merges, by adding them together.

For a probabilistic `pif`-statement, we similarly need to define the operation of the probabilities on the measure. Rather than separating the measure into two parts, however, its effect is to *scale* the total weight. This is illustrated in Figure 3.4, where a measure is shown figuratively as a continuous density function over a single variable. In both cases, the sum of the measures on each branch is equal to the original measure. Using the operator $e_q(\mu)(X) = q.\mu(X)$, where $q \in [0,1]$ is a probability, the semantics of probabilistic choice is as follows:

$$[\![O.f \mid \texttt{pif}\ q\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]_p = [\![O.f \mid C_1]\!]_p \circ e_q + [\![O.f \mid C_2]\!]_p \circ e_{1-q}$$

The semantics of `skip` is simply the identity map on measures — $\lambda\mu.\mu$ — and that of a `return` statement occurring in a method $O.f$ is as follows:

$$[\![O.f \mid \texttt{return}\ E]\!]_p(\mu) = [\![O.f \mid X_{O.f} := E]\!]_p$$

We will not deal with remote method calls at this moment, and so we only have local method calls left to consider. In this case, we can perform a direct substitution as follows, where the called method is associated with the same object O as the callee:

$$\llbracket O.f \mid X_i := f'(X_{i_1},\dots,X_{i_n}) \rrbracket_p = \\ \llbracket O.f' \mid C\,\{X_{i_1}/X_{j_1},\dots,X_{i_n}/X_{j_n}, X_i := E/\texttt{return}\,E\} \rrbracket_p$$

where $\mathit{def}(O.f') = O.f'(X_{j_1},\dots,X_{j_n})\ \{C\}$. This can be thought of as modifying the denotation of the body of the method so that its argument variables $X_{j_1},\dots,X_{j_n}$ are replaced by the actual arguments $X_{i_1},\dots,X_{i_n}$, and the `return` call is replaced by a direct assignment to the variable X_i.

Theorem 3.2.1. *For a* Siril *command C, occurring in a method $O.f$ containing N variables, the denotational semantics $\llbracket O.f \mid C \rrbracket_p$ describes a continuous linear operator on the Banach space $(\boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)}), \|\cdot\|)$.*

Proof: To prove linearity, we need to show that for all measures μ, μ_1, μ_2 and constants $c \in \mathbb{R}$:

$$\begin{aligned} \llbracket O.f \mid C \rrbracket_p(c\mu) &= c\llbracket O.f \mid C \rrbracket_p(\mu) \\ \llbracket O.f \mid C \rrbracket_p(\mu_1 + \mu_2) &= \llbracket O.f \mid C \rrbracket_p(\mu_1) + \llbracket O.f \mid C \rrbracket_p(\mu_2) \end{aligned}$$

Since the composition of two linear operators is linear, as is the sum of two linear operators, we just need to show this for the non-trivial base operators: $\llbracket O.f \mid X_i := E \rrbracket_p$, $e_{\llbracket B \rrbracket}$, and e_q. In the first case, we have (from the definition of point-wise addition and scalar multiplication for measures):

$$\begin{aligned} \llbracket O.f \mid X_i := E \rrbracket_p(c\mu) &= (c\mu) \circ \llbracket X_i := E \rrbracket^{-1} \\ &= c(\mu \circ \llbracket X_i := E \rrbracket^{-1}) \\ &= c\llbracket O.f \mid X_i := E \rrbracket_p(\mu) \end{aligned}$$

$$\begin{aligned} \llbracket O.f \mid X_i := E \rrbracket_p(\mu_1 + \mu_2) &= (\mu_1 + \mu_2) \circ \llbracket X_i := E \rrbracket^{-1} \\ &= \mu_1 \circ \llbracket X_i := E \rrbracket^{-1} + \mu_2 \circ \llbracket X_i := E \rrbracket^{-1} \\ &= \llbracket O.f \mid X_i := E \rrbracket_p(\mu_1) + \llbracket O.f \mid X_i := E \rrbracket_p(\mu_2) \end{aligned}$$

It is straightforward to show the linearity of $e_{\llbracket B \rrbracket}$, and e_q by similar arguments.

The continuity of the semantics can be shown similarly — the composition and sum of two continuous functions is continuous, and so we just need to show the continuity of $\llbracket O.f \mid X_i := E \rrbracket_p$, $e_{\llbracket B \rrbracket}$, and e_q, which follows trivially from their definitions. ■

As a small example of the denotational semantics of Siril, consider the following method, which has only a single variable X:

```
O.f(X) {
  if (0.5) then
    X := X + 1
  else
    X := X - 1;
  return X
}
```

Its semantics is given by:

$$\begin{aligned}
[\![O.f]\!]_p(\mu) &= ([\![O.f \mid X := X+1]\!]_p \circ e_{0.5} + [\![O.f \mid X := X-1]\!]_p \circ e_{0.5})(\mu) \\
&= ([\![O.f \mid X := X+1]\!]_p \circ e_{0.5})(\mu) + ([\![O.f \mid X := X-1]\!]_p \circ e_{0.5})(\mu) \\
&= [\![O.f \mid X := X+1]\!]_p(0.5\,\mu) + [\![O.f \mid X := X-1]\!]_p(0.5\,\mu) \\
&= 0.5\,\mu \circ [\![X := X+1]\!]^{-1} + 0.5\,\mu \circ [\![X := X-1]\!]^{-1}
\end{aligned}$$

Hence if the initial measure μ gives a uniform distribution over $[1,3]$ of the value of X, then $[\![O.f]\!]_p(\mu)$ gives a uniform distribution over $[0,2]$ with probability 0.5, and over $[2,4]$ with probability 0.5, which is the same as a uniform distribution over $[0,4]$.

Since Siril does not contain loops, this denotational semantics can be considered a subset of that presented in [112]. For a discussion of how to extend the language and its semantics with loops, see Appendix A. Note that there are parallels between this sequential fragment of Siril and the probabilistic guarded constraint language (pGCL) [124], except that in the latter, a weakest precondition approach is taken (as opposed to our strongest postcondition approach), and only discrete distributions are considered (as opposed to continuous probability measures).

3.3 Probabilistic Semantics of Siril

The semantics that we just described is a denotational one, and works well for single-threaded programs. There has been work on denotational semantics for languages with probabilistic and concurrent features [134], but the problem with such an approach is that it describes the entire program as a *single* operator. This means that we lose information about the structure of the program (e.g. the distribution of computation into threads), and makes describing non-determinism more complicated. In our case, we would like to maintain some information about the control flow structure, which will ultimately enable us to construct *compositional* performance models from Siril programs in the next chapter.

Historically, *operational semantics* have been used to describe concurrency, since the non-determinism of interleaved execution maps directly to non-determinism between possible transitions in the semantics. In our case, we introduce a *probabilistic automaton semantics*[2], which essentially adds a control-flow graph structure to Kozen's semantics. This is a novel approach, in which we combine Kozen's denotational semantics with an automaton structure. In this section, we define this semantics, $[\![\cdot]\!]_{pa}$, for the full language Siril[3].

In addition to the linear operators of Kozen's semantics, we introduce an *automaton* for each method in a Siril program. We will refer to the states of the automaton as *stages*, to avoid confusion when we talk about states of the program — essentially, we use the term *stage* for a program point (a value of the program counter), and *state* for a program point along with a measure over the values of the variables in the program. A *transition* between two stages is labelled with a linear operator, which corresponds to Kozen's denotational semantics $[\![\cdot]\!]_p$ over sequential fragments. We only introduce stages when there is a remote procedure call, and the method we call is recorded as a label on the stage. Branches due to conditional statements do not require additional stages, and can be represented using multiple transitions between stages (as opposed to addition in the denotational semantics $[\![\cdot]\!]_p$).

Formally, the semantics of a Siril program consists of the following elements:

1. A vector $\boldsymbol{X}$ of N distinct variables, $X_1,\dots,X_N$, with $\boldsymbol{X}(i)$ denoting the ith variable. We use $\iota_{\boldsymbol{X}}(X)$ to denote the index of variable X in $\boldsymbol{X}$, such that $\boldsymbol{X}(\iota_{\boldsymbol{X}}(X)) = X$ holds[4].

2. A vector $\boldsymbol{O}$ of M distinct objects, $O_1,\dots,O_M$. These are fixed, in the sense that there is no creation or deletion of objects.

3. A set of methods, $\mathcal{F}(O_i)$, for each object O_i.

4. An automaton $[\![O.f]\!]_{pa} = (\mathcal{S},\mathcal{T})$ for each method $O.f$. Here, $\mathcal{S}$ is a set of stages (the states of the automaton), and $\mathcal{T} \subseteq \mathcal{S}\times\mathcal{O}\times\mathcal{S}$ is a transition relation, where $\mathcal{O}$

[2] This is not to be confused with the classical notion of a probabilistic automaton [147], where transitions are labelled by probabilities. In our case, the transitions are labelled by operators on measures.

[3] Note that if we were to simplify our semantics to the non-probabilistic case — i.e. where all the measures are point measures — this would correspond to a deterministic denotational semantics embedded on the control flow graph of the program, in the absence of the probabilistic conditional (`pif`) command.

[4] We introduce the function ι_X here for completeness — we do not need to make use of it until constructing our abstract collecting semantics in the next chapter (Section 4.5).

is the set of linear operators $M\colon \boldsymbol{B} \to \boldsymbol{B}$ on measures μ in $\boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$. We will write a transition $(s, M, s') \in \mathcal{T}$ as $s \xrightarrow{M} s'$, for convenience.

We will define the automaton semantics of a method $O.f$ in two parts, as follows:

$$[\![O.f]\!]_{pa} = \left([\![O.f]\!]^S_{pa}, [\![O.f]\!]^T_{pa}\right)$$

If the method is defined such that $def(O.f) = O.f(X_1, \ldots, X_n)\{C\}$, then the stages and transitions are as follows:

$$\begin{aligned} [\![O.f]\!]^S_{pa} &= \left\{\circ_{O.f}, \bullet_{O.f}\right\} \cup [\![O.f \mid C]\!]^S_{pa} \\ [\![O.f]\!]^T_{pa} &= [\![O.f \mid C]\!]^T_{pa} \end{aligned}$$

Here, the stages $\circ_{O.f}$ and $\bullet_{O.f}$ are the respectively the entry and exit points in the automaton for the method $O.f$. We call these *external stages*. When the method name is clear from context, we will omit the subscript $O.f$. Stages $s \in [\![O.f \mid C]\!]^S_{pa}$ are called *internal stages*, and have a label $L \in \{O_i.f \mid 1 \le i \le M, f \in \mathcal{F}(O_i)\}$. We use the internal stages to represent calls to other methods (i.e. remote procedure calls), and their label records the method that is called. We write $s[L]$ for an internal stage s with label L.

Since the variables, objects, and methods in a Siril program can be easily determined statically, we will assume in the following that $\boldsymbol{X}$, $\boldsymbol{O}$ and $\mathcal{F}$ are known and fixed. In particular, this means that we assume that there are no name conflicts between variables in different methods — a condition easily met by replacing a method with an alpha-equivalent version.

We can now describe our probabilistic automaton semantics over Siril commands. We begin with the `skip` command, which is trivially the identity map ($I = \lambda\mu.\ \mu$) on the input measure, and has no internal stages:

$$[\![O.f \mid \texttt{skip}]\!]^S_{pa} = \{\,\} \qquad [\![O.f \mid \texttt{skip}]\!]^T_{pa} = \{\circ \xrightarrow{I} \bullet\}$$

For `return` statements, we need to assign the value of the expression to the correct return variable — but this assignment is precisely the probabilistic semantics, $[\![O.f \mid \texttt{return}\ E]\!]_p$. As with `skip`, no internal stages are added.

$$\begin{aligned} [\![O.f \mid \texttt{return}\ E]\!]^S_{pa} &= \{\,\} \\ [\![O.f \mid \texttt{return}\ E]\!]^T_{pa} &= \{\circ \xrightarrow{M} \bullet\} \text{ where } M = [\![O.f \mid \texttt{return}\ E]\!]_p \end{aligned}$$

Basic assignments do not introduce any internal stages, and are denoted by the same operator as in Kozen's semantics:

$$\begin{aligned} [\![O.f \mid X_i := E]\!]^S_{pa} &= \{\,\} \\ [\![O.f \mid X_i := E]\!]^T_{pa} &= \{\circ \xrightarrow{M} \bullet\} \text{ where } M = [\![X_i := E]\!]_p \end{aligned}$$

The semantics of calling a local method f', defined as $O.f'(X_{j_1},\ldots,X_{j_n})\ \{C\}$, from within a method $O.f$ is similar to the probabilistic case — we substitute the argument variables of the method for the arguments of the call, and actually perform the assignment of the return variable for each `return` command:

$$[\![O.f \mid X_i := f'(X_{i_1},\ldots,X_{i_n})]\!]_{pa} = \\ [\![O.f' \mid C\,\{X_{i_1}/X_{j_1},\ldots,X_{i_n}/X_{j_n}, X_i := E/\texttt{return}\ E\}]\!]_{pa}$$

Sequencing involves composing two automata, so that the exit transitions of the first are merged with the entry transitions of the second. The resulting linear operator is the composition of the original operators. If there is more than one start or exit transition, we must take all possible combinations. Hence in the worst case, the number of transitions will grow exponentially in the number of branching instructions. Formally, the semantics is as follows:

$$\begin{array}{rcl}
[\![O.f \mid C_1\ ;\ C_2]\!]^S_{pa} & = & [\![O.f \mid C_1]\!]^S_{pa} \cup [\![O.f \mid C_2]\!]^S_{pa} \\
[\![O.f \mid C_1\ ;\ C_2]\!]^T_{pa} & = & \{s \xrightarrow{M_2 \circ M_1} s' \mid s \xrightarrow{M_1} \bullet \in [\![O.f \mid C_1]\!]^T_{pa} \wedge \circ \xrightarrow{M_2} s' \in [\![O.f \mid C_2]\!]^T_{pa}\} \cup \\
& & \{s \xrightarrow{M} s' \in [\![O.f \mid C_1]\!]^T_{pa} \mid s' \neq \bullet\} \cup \\
& & \{s \xrightarrow{M} s' \in [\![O.f \mid C_2]\!]^T_{pa} \mid s \neq \circ\}
\end{array}$$

For conditional statements, we again employ Kozen's semantics, except that instead of adding the measures from each branch, we introduce separate transitions in the automaton. This has the advantage (from the point of view of our analysis) of limiting the behaviour of the operators on measures, at the expense of an exponential blow-up in the number of transitions when there is a sequence of conditional statements.

The semantics of the `if`- and `pif`-statements are as follows:

$$\begin{array}{rcl}
[\![O.f \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]^S_{pa} & = & [\![O.f \mid C_1]\!]^S_{pa} \cup [\![O.f \mid C_2]\!]^S_{pa} \\
[\![O.f \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]^T_{pa} & = & \{\circ \xrightarrow{M \circ e_{[\![B]\!]}} s \mid \circ \xrightarrow{M} s \in [\![O.f \mid C_1]\!]^T_{pa}\} \cup \\
& & \{s \xrightarrow{M} s' \in [\![O.f \mid C_1]\!]^T_{pa} \mid s \neq \circ\} \cup \\
& & \{\circ \xrightarrow{M \circ e_{[\![\neg B]\!]}} s \mid \circ \xrightarrow{M} s \in [\![O.f \mid C_2]\!]^T_{pa}\} \cup \\
& & \{s \xrightarrow{M} s' \in [\![O.f \mid C_2]\!]^T_{pa} \mid s \neq \circ\}
\end{array}$$

$$\begin{array}{rcl}
[\![O.f \mid \texttt{pif}\ q\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]^S_{pa} & = & [\![O.f \mid C_1]\!]^S_{pa} \cup [\![O.f \mid C_2]\!]^S_{pa} \\
[\![O.f \mid \texttt{pif}\ q\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]^T_{pa} & = & \{\circ \xrightarrow{M \circ e_q} s \mid \circ \xrightarrow{M} s \in [\![O.f \mid C_1]\!]^T_{pa}\} \cup \\
& & \{s \xrightarrow{M} s' \in [\![O.f \mid C_1]\!]^T_{pa} \mid s \neq \circ\} \cup \\
& & \{\circ \xrightarrow{M \circ e_{1-q}} s \mid \circ \xrightarrow{M} s \in [\![O.f \mid C_2]\!]^T_{pa}\} \cup \\
& & \{s \xrightarrow{M} s' \in [\![O.f \mid C_2]\!]^T_{pa} \mid s \neq \circ\}
\end{array}$$

Finally, we consider the semantics of remote procedure calls. Consider a call to a method $O'.f'$ such that $def(O'.f') = O'.f'(X_{j_1},\ldots,X_{j_n})$ (i.e. $X_{j_1},\ldots,X_{j_n}$ are the argument variables of $O'.f'$). Then the semantics is as follows, where s^* is a fresh stage:

$$\begin{aligned} [\![O.f \mid X_i := O'.f'(X_{i_1},\ldots,X_{i_n})]\!]^S_{pa} &= \{s^*[O'.f']\} \\ [\![O.f \mid X_i := O'.f'(X_{i_1},\ldots,X_{i_n})]\!]^T_{pa} &= \{\circ \xrightarrow{M_{in}} s^*[O'.f'], s^*[O'.f'] \xrightarrow{M_{out}} \bullet\} \end{aligned} \tag{3.1}$$

where M_{in} and M_{out} perform the copying between the local variables of $O.f$ and the return and argument variables of $O'.f'$:

$$\begin{aligned} M_{in} &= [\![X_{j_1} := X_{i_1}]\!]_p \circ \cdots \circ [\![X_{j_n} := X_{i_n}]\!]_p \\ M_{out} &= [\![X_i := X_{O'.f'}]\!]_p \end{aligned}$$

It should be noted that our probabilistic automaton semantics of SIRIL is not *fully abstract*, in the sense of two programs that are observably equivalent having the same semantics. As an example, consider a program that consists of two syntactically identical methods, $O_1.f_1$ and $O_1.f_2$, and a third method $O_2.f$ that calls both of them sequentially. Behaviourally, the program is the same as one where $O_2.f$ calls $O_1.f_1$ twice, but the probabilistic automaton semantics differ, because the labels of the stages will be different.

3.3.1 An Example of the Automaton Semantics of SIRIL

Figure 3.5 shows the concrete semantics of the `Client.buy` method from Figure 3.1, as an example. The program is repeated alongside its semantics, with the variables shortened for brevity — `quantity` becoming Q, `cost` becoming C, and so forth. There are two internal stages, corresponding to the two remote procedure calls. We use $arg_{S.P}$ and $arg_{S.B}$ in place of the actual argument variables of `Server.getPrice` and `Server.buy` respectively. The return variables are $X_{C.B}$, $X_{S.P}$, and $X_{S.B}$ for `Client.buy`, `Server.getPrice`, and `Server.buy` respectively.

To "execute" a method's probabilistic automaton semantics, we need to supply the initial measure over its variables — that is to say, all of its variables and not just its arguments. A sensible approach is to initialise all the non-arguments to the point measure zero. But what does an execution of a method actually look like in this semantics? And more importantly, how do we handle the distributed nature of the objects, which can be thought of as running in parallel? We will answer these questions by defining a *probabilistic interpretation* of our semantics.

```
Client.buy(Q, C) {
  P := Server.getPrice(Q);
  if (P ≤ C) then
    S := Server.buy(Q);
    if (S = 1) then
      C := C − P;
  return C
}
```

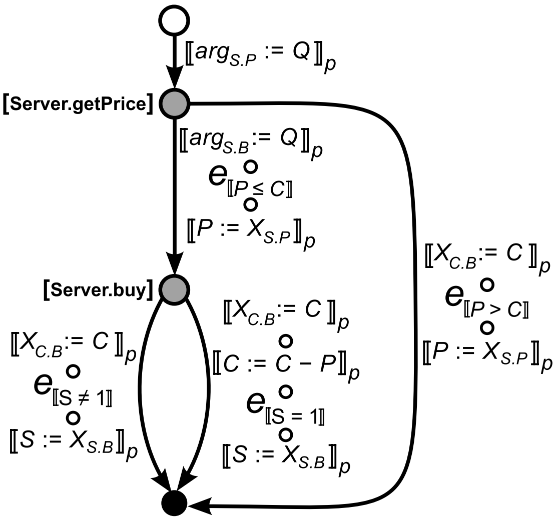


Figure 3.5: Concrete semantics of the `Client.buy` method from Figure 3.1

3.4 Probabilistic Interpretation of Siril

The probabilistic automaton semantics encapsulates a program's behaviour under any input measure, but if we execute the program given a *particular* input measure, we expect to see a particular probabilistic behaviour. When a stage in the probabilistic automaton has multiple outgoing transitions, we can envisage the measure splitting apart and flowing down each transition simultaneously. There is uncertainty about which transition to take due to uncertainty about the values of the variables.

Let us consider a Siril method $O.f$ that contains no remote procedure calls. For each stage $s \in [\![O.f]\!]^S_{pa}$, there are a number of transitions of the form $s \xrightarrow{M} s'$ that can be taken from it. For a particular transition $s \xrightarrow{M} s'$, and a particular starting measure μ, the probability of moving to s' is given by:

$$p = \frac{(M(\mu))(\mathbb{R}^N)}{\mu(\mathbb{R}^N)}$$

In other words, the probability is the ratio between the total weight before and after taking the transition. This result depends on the particular semantics for Siril that we presented in the previous section — if we were to allow arbitrary measure operators M on transitions, it would not necessarily hold. Our semantics ensures this because the transitions out of a stage preserve the total weight of the measure that entered it. This is stated more formally in the following theorem.

Theorem 3.4.1. *For every stage $s \in [\![O.f]\!]^S_{pa}$ in the probabilistic automaton semantics of a* Siril *method $O.f$, and every measure $\mu \in \boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$, the following conservation* *law holds:*

$$\sum_{\{ s_1 \xrightarrow{M} s_2 \in [\![O.f]\!]^T_{pa} \mid s_1 = s \}} (M(\mu))(\mathbb{R}^N) = \mu(\mathbb{R}^N)$$

Furthermore, the set $\{ s_1 \xrightarrow{M} s_2 \in [\![O.f]\!]^T_{pa} \mid s_1 = s \}$ *is finite.*

To prove this, we first need to establish the following two lemmas:

Lemma 3.4.2. *For all measures* $\mu \in \boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$*, and all* Siril *arithmetic expressions* E*:*

$$[\![O.f \mid X_i := E]\!]_p(\mu)(\mathbb{R}^N) = \mu(\mathbb{R}^N)$$

Proof: Let us write $[\![X_i := E]\!]_p$ in place of $[\![O.f \mid X_i := E]\!]_p$, as the method name is unimportant in this context. Since $[\![X_i := E]\!]$ is a total function, its domain is precisely $\mathbb{R}^N$. Hence the codomain of $[\![X_i := E]\!]^{-1}$ is also $\mathbb{R}^N$, which means that $[\![X_i := E]\!]^{-1}(\mathbb{R}^N) = \mathbb{R}^N$. It follows that $[\![X_i := E]\!]_p(\mu)(\mathbb{R}^N) = \mu \circ [\![X_i := E]\!]^{-1}(\mathbb{R}^N) = \mu(\mathbb{R}^N)$. ■

Lemma 3.4.3. *For all measures* $\mu \in \boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$*, and all* Siril *Boolean expressions* B *and probabilities* $q \in [0,1]$*:*

$$\begin{array}{lcl} e_{[\![B]\!]}(\mu)(\mathbb{R}^N) + e_{[\![\neg B]\!]}(\mu)(\mathbb{R}^N) & = & \mu(\mathbb{R}^N) \\ e_q(\mu)(\mathbb{R}^N) + e_{1-q}(\mu)(\mathbb{R}^N) & = & \mu(\mathbb{R}^N) \end{array}$$

Proof: The first case can be proven by the following argument:

$$\begin{array}{lcl} e_{[\![B]\!]}(\mu)(\mathbb{R}^N) + e_{[\![\neg B]\!]}(\mu)(\mathbb{R}^N) & = & \mu([\![B]\!] \cap \mathbb{R}^N) + \mu([\![\neg B]\!] \cap \mathbb{R}^N) \\ & = & \mu([\![B]\!] \cap \mathbb{R}^N) + \mu([\![B]\!]^C \cap \mathbb{R}^N) \\ & = & \mu(([\![B]\!] \cap \mathbb{R}^N) \cup ([\![B]\!]^C \cap \mathbb{R}^N)) \\ & = & \mu(\mathbb{R}^N) \end{array}$$

where $[\![B]\!]^C$ is the complement of the set $[\![B]\!]$. The second case follows trivially from the definition of e_q. ■

Proof: [Theorem 3.4.1] The proof proceeds by structural induction on the body of $O.f$. In the base cases of `skip`, `return`, assignment, and remote procedure calls, only stages with a single outgoing transition are introduced, and such a transition is labelled with either the identity map or an assignment map $[\![X_i := E]\!]_p$. In the former, the total weight of the measure is trivially preserved, and in the latter, it is preserved due to Lemma 3.4.2. There are two inductive cases:

1. In the case of a sequential composition 'C_1 ; C_2', we combine the two sets of transitions $[\![O.f \mid C_1]\!]^T_{pa}$ and $[\![O.f \mid C_2]\!]^T_{pa}$. We will consider just two subsets,

$T_1 = \{\, s_i \xrightarrow{M_i} \bullet \mid 1 \leq i \leq n \} \subseteq \llbracket O.f \mid C_1 \rrbracket^T_{pa}$ and $T_2 = \{ \circ \xrightarrow{M_j} s'_j \mid 1 \leq j \leq m \} \subseteq \llbracket O.f \mid C_2 \rrbracket^T_{pa}$, since most of the transitions remain unchanged when we compose the two automata. The semantics of sequential composition results in a new set of transitions $T' = \{\, s_i \xrightarrow{M_j \circ M_i} s'_j \mid 1 \leq i \leq n, 1 \leq j \leq m \} \subseteq \llbracket O.f \mid C_1 \; ; \; C_2 \rrbracket^T_{pa}$. Given that the transitions in $\llbracket O.f \mid C_1 \rrbracket^T_{pa}$ and $\llbracket O.f \mid C_2 \rrbracket^T_{pa}$ preserve the total weight by the induction hypothesis, we can show the following for each stage s for which there is a transition in T_1, where we omit the method name $O.f$, and write $\{\, s \xrightarrow{M} s_2 \in T \,\}$ as shorthand for $\{\, s_1 \xrightarrow{M} s_2 \in T \;\mid\; s_1 = s \}$:

$$\begin{aligned}
&\sum_{\{\, s \xrightarrow{M} s_2 \in \llbracket C_1 \; ; \; C_2 \rrbracket^T_{pa} \}} (M(\mu))(\mathbb{R}^N) \\
= &\sum_{\{\, s \xrightarrow{M} s_2 \in \llbracket C_1 \; ; \; C_2 \rrbracket^T_{pa} \setminus T' \}} (M(\mu))(\mathbb{R}^N) + \sum_{\{\, s \xrightarrow{M} s_2 \in T' \}} (M(\mu))(\mathbb{R}^N) \\
= &\sum_{\{\, s \xrightarrow{M} s_2 \in \llbracket C_1 \rrbracket^T_{pa} \mid s_2 \neq \bullet \}} (M(\mu))(\mathbb{R}^N) + \sum_{\{\, s \xrightarrow{M} \bullet \in T_1 \}} \sum_{\{ \circ \xrightarrow{M'} s_2 \in T_2 \}} (M' \circ M(\mu))(\mathbb{R}^N) \\
= &\sum_{\{\, s \xrightarrow{M} s_2 \in \llbracket C_1 \rrbracket^T_{pa} \mid s_2 \neq \bullet \}} (M(\mu))(\mathbb{R}^N) + \sum_{\{\, s \xrightarrow{M} \bullet \in T_1 \}} (M(\mu))(\mathbb{R}^N) \\
= &\sum_{\{\, s \xrightarrow{M} s_2 \in \llbracket C_1 \rrbracket^T_{pa} \}} (M(\mu))(\mathbb{R}^N) \\
= &\; \mu(\mathbb{R}^N)
\end{aligned}$$

If a stage does not have a transition in T_1, its transitions are unmodified in $\llbracket O.f \mid C_1 \; ; \; C_2 \rrbracket^T_{pa}$. Hence sequential composition preserves the total weight of a measure.

2. For conditional branching, we take each existing transition out of a stage s, and replace it with two new transitions $s \xrightarrow{M_1 \circ M} s'_1$ and $s \xrightarrow{M_2 \circ M} s'_2$, where M_1 and M_2 are of the form $e_{\llbracket B \rrbracket}$ and $e_{\llbracket \neg B \rrbracket}$ respectively, or else are e_q and e_{1-q}. Since, by Lemma 3.4.3, $M_1(M(\mu))(\mathbb{R}^N) + M_2(M(\mu))(\mathbb{R}^N) = M(\mu)(\mathbb{R}^N)$, the new transitions preserve the total weight of the original measure.

■

Formally, the probabilistic interpretation of a Siril program is an *acyclic DTMC*, induced by a transition system over states of the form $\mu \vdash s$. If we consider a single method $O.f$, $s \in \llbracket O.f \rrbracket^S_{pa}$ is a stage in its automaton, and $\mu \in \boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$ is a measure over the state of its variables. If $O.f$ contains no remote procedure calls, its probabilistic transition system is defined as follows:

$$\mu \vdash s \xrightarrow{p} \mu' \vdash s' \;\text{ iff }\; s \xrightarrow{M} s' \in \llbracket O.f \rrbracket^T_{pa} \wedge \mu' = M(\mu) \tag{3.2}$$

where $p = \frac{(M(\mu))(\mathbb{R}^N)}{\mu(\mathbb{R}^N)}$. Theorem 3.4.1 ensures that the probabilities of the transitions out of a state $\mu \vdash s$ sum to one. The interpretation $[\![O.f]\!]_{pa}(\mu)$ is the probabilistic graph induced by $\mu \vdash \bigcirc_{O.f}$.

Of course, we are really interested in the behaviour of multi-threaded systems, in which remote procedure calls occur. In this case all of the objects in the SIRIL program exist in parallel[5], with the ability to invoke one another's methods. Recall that an instantiation of a program is a call to a particular method $O.f(X_1,\ldots,X_n)$. Since we want a probabilistic interpretation, we need to provide an initial probability measure μ_I that gives the distribution of the inputs $X_1,\ldots,X_n$ (we will worry about how to actually specify such a measure in the next chapter).

Consider the following system, in which we can think of all the methods of all the objects in a SIRIL program running in parallel:

$$O_1.f_1 \parallel \cdots \parallel O_1.f_{n_1} \parallel \cdots \parallel O_M.f_1 \parallel \cdots \parallel O_M.f_{n_M} \tag{3.3}$$

Initially, only the instantiated method $O.f$ is actively executing, and the others are inactive. When a remote procedure call takes place, $O.f$ blocks and another method is activated. Importantly, only *one* method is ever active in our system at any one time, in the sense that all other methods must be either blocked or not instantiated. In other words, there is a *single* thread of execution, which transfers control to different methods by remote invocation.

Note that from a performance analysis point of view, we are more interested in the case when many methods are executing at the same time, and there is contention for resources — for example, if there are many clients that want to communicate with a server at the same time. We will look at this sort of problem in more detail in the next chapter, when we show how to construct PEPA models from SIRIL programs. For now, however, we should bear in mind that each object represents a single resource. By placing all its methods in parallel, we do not mean that they can be called at the *same* time — we are merely using this structure to interpret the *probabilistic* behaviour of the system. Another way to think about it is that we are analysing the system from the perspective of an individual user, and we will later combine such analyses to build a performance model of the system with multiple users.

There are two levels at which we can construct our probabilistic interpretation:

[5] Recall that objects are static entities that correspond to resources, or locations. Objects are neither created nor destroyed — they exist for the entire lifetime of the program. The methods of an object, however, must be invoked in order to execute.

1. As a *discrete time interpretation*, where we consider only the probabilistic behaviour of the system. From this we can construct a *discrete time Markov chain* (DTMC). We will describe this in Section 3.4.1.

2. As a *continuous time interpretation*, where events such as remote procedure calls also have a duration. If the durations of transitions are exponentially distributed, this leads to a *continuous time Markov chain* (CTMC). We will describe this in Section 3.4.2.

3.4.1 Discrete Time Interpretation

If we have no information about how long it takes for instructions to execute and remote objects to be invoked, we can construct a *time-abstract* interpretation of a Siril program — we can determine the probability of evolving to a new state, but not how long it takes for this to happen. States in the interpretation are of the form $\mu \vdash s_{1,1} \parallel \cdots \parallel s_{M,n_M}$, where for every method $O_i.f_j$ of every object, $s_{i,j}$ denotes the current stage in the method's probabilistic automaton semantics — i.e. $s_{i,j} \in [\![O_i.f_j]\!]^S_{pa}$. If we instantiate the program with a call to the method $O_i.f_j$, and the initial measure over the variables is μ_I, then the initial state in the probabilistic interpretation is:

$$\mu_I \vdash \bullet_{O_1.f_1} \parallel \cdots \parallel \bullet_{O_i.f_{j-1}} \parallel \circ_{O_i.f_j} \parallel \bullet_{O_i.f_{j+1}} \parallel \cdots \parallel \bullet_{O_M.f_{n_M}} \tag{3.4}$$

In other words, all methods begin in state $\bullet$ except for the method we instantiate, $O_i.f_j$, which starts in state $\circ$.

The transitions out of each state $\mu \vdash s_{1,1} \parallel \cdots \parallel s_{M,n_M}$ are as follows, and are labelled with a probability p:

$$\begin{aligned}\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{M,n_M} \xrightarrow{p} \mu' \vdash s_{1,1} \parallel \cdots \parallel \bullet_{O_i.f_j} \parallel \cdots \parallel s_{M,n_M}& \\ \text{iff } \mu \vdash s_{i,j} \xrightarrow{p} \mu' \vdash \bullet_{O_i.f_j} \wedge \neg \mathit{blocked}(s_{i,j})&\end{aligned} \tag{3.5}$$

$$\begin{aligned}\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{i',j'} \parallel \cdots \parallel s_{M,n_M} \xrightarrow{p} \mu' \vdash s_{1,1} \parallel \cdots \parallel s'_{i,j} \parallel \cdots \parallel \circ_{O_{i'}.f_{j'}} \parallel \cdots \parallel s_{M,n_M}& \\ \text{iff } \mu \vdash s_{i,j} \xrightarrow{p} \mu' \vdash s'_{i,j} \wedge s'_{i,j} = s[O_{i'}.f_{j'}] \wedge \neg \mathit{blocked}(s_{i,j})&\end{aligned} \tag{3.6}$$

Here, the first case corresponds to returning from a remote procedure call (i.e. entering the $\bullet$ stage), and the second corresponds to invoking such a call (i.e. causing another method to enter its $\circ$ stage). For this to make sense, we initially require that all methods except the one invoked are in the ● stage. Vitally, we use the predicate *blocked*$(s_{i,j})$ to describe when a method is unable to execute:

$$blocked(s_{i,j}) \text{ iff } s_{i,j} = \bullet_{O_i.f_j} \vee (s_{i,j} = s[O_{i'}.f_{j'}] \wedge s_{i',j'} \neq \bullet_{O_{i'}.f_{j'}}) \tag{3.7}$$

Informally, a method is in a blocked stage if it has finished executing (i.e. it is in its ● stage), or it is in the middle of a remote procedure call (i.e. the method it is calling is not in its ● stage).

For the interpretation to give rise to a DTMC, starting in the initial state shown in Equation 3.4, we need to ensure that only one method is enabled at a time. If this were not the case, the sum of the probabilities on the transitions out of each state might be greater than one, and would not describe a probability distribution. This singular enabling of methods is given by the following theorem:

Theorem 3.4.4. *Starting in a state $\mu \vdash s_{1,1} \parallel \cdots \parallel s_{M,n_M}$ where $\neg blocked(s_{i,j})$ holds of precisely one $s_{i,j}$, it is not possible to reach a state in which $\neg blocked(s_{i,j})$ holds of more than one $s_{i,j}$.*

Proof: The proof follows from the definition of $\xrightarrow{p}$. In the first case, we finish executing a method, entering the ● stage so that it becomes blocked. This causes at most one other method to become unblocked — otherwise, this would mean that we were called by more than one method, which is not possible in the absence of recursion. In the second case, we invoke a remote procedure call, thus unblocking one method but in the process becoming blocked ourselves. Hence the number of blocked methods in the system is unchanged. ■

3.4.2 Continuous Time Interpretation

If we are to analyse the *performance* of a Siril program, it is essential to have a less abstract notion of time. This is not an inherent property of the program, since it depends entirely on the execution architecture and the characteristics of the network. Because of this, we will introduce timing information at the semantic level, in that we use a random variable to describe the time taken to perform each ‘operation’. In our context, an operation is either an internal sequence of computations, corresponding to a transition in the probabilistic automaton, or the invocation of a remote procedure call.

There are various granularities at which we could choose to define the duration of operations, but to simplify our presentation we will assign just two random variables to each object O — one corresponding to internal operations, and the other corresponding to network communication. Furthermore, in order to generate a Markov chain, we will consider only exponentially distributed random variables $X \sim Exp(r)$ — namely, $\mathbf{Pr}(X < t) = 1 - e^{-rt}$ for all times t. This means that we can represent X by its parameter r as follows:

- $R_I(O) \in \mathbb{R}^+$ is the *internal rate* of the object O.
- $R_E(O) \in \mathbb{R}^+$ is the *external rate* of the object O.

This assumption of an exponential distribution is important if we are to construct a feasible analysis of the performance of a Siril program. In essence, it allows us to correctly capture the mean duration of an operation, but not its variance, and so should be seen as an approximation. We could extend our technique to deal with general distributions, but this would not result in a Markov chain, and so we would have to resort to simulation as an analyse technique.

For a method $O.f$ that contains no remote procedure calls, our continuous time interpretation results in a rate-labelled transition system as follows:

$$\mu \vdash s \xrightarrow{r} \mu' \vdash s' \quad \text{iff} \quad \mu \vdash s \xrightarrow{p} \mu' \vdash s' \wedge r = p.R_I(O)$$

In this case, only the internal rate of the method is needed, but we need to be a little more careful when there are remote procedure calls.

Consider a system $S = s_{1,1} \parallel \cdots \parallel s_{M,n_M}$. We can observe from the definition of $\xrightarrow{p}$ that whenever $\mu \vdash S \xrightarrow{p} \mu' \vdash S'$, this corresponds either to invoking a remote procedure call or returning from one. The operation leading to this call or return is internal, but it is followed by some communication over the network. We therefore need to include the external rate in our rate-labelled transition system:

$$\begin{aligned}&\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{M,n_M} \xrightarrow{r} \mu' \vdash s_{1,1} \parallel \cdots \parallel \bullet_{O_i.f_j} \parallel \cdots \parallel s_{M,n_M} \\ &\quad \text{iff } \mu \vdash s_{i,j} \xrightarrow{p} \mu' \vdash \bullet_{O_i.f_j} \wedge \neg blocked(s_{i,j}) \wedge r = p.((R_I(O_i))^{-1} + (R_E(O_i))^{-1})^{-1}\end{aligned} \tag{3.8}$$

$$\begin{aligned}&\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{i',j'} \parallel \cdots \parallel s_{M,n_M} \xrightarrow{r} \mu' \vdash s_{1,1} \parallel \cdots \parallel s'_{i,j} \parallel \cdots \parallel \circ_{O_{i'}.f_{j'}} \parallel \cdots \parallel s_{M,n_M} \\ &\text{iff } \mu \vdash s_{i,j} \xrightarrow{p} \mu' \vdash s'_{i,j} \wedge s'_{i,j} = s[O_{i'}.f_{j'}] \wedge \neg blocked(s_{i,j}) \wedge r = p.((R_I(O_i))^{-1} + (R_E(O_{i'}))^{-1})^{-1}\end{aligned} \tag{3.9}$$

The value of the rate r is calculated as the probability of the transition, divided by the expected duration of the transition (which is the sum of the inverses of the internal and external rates). This is technically an approximation, since the convolution of two exponentials is not itself exponentially distributed. However, since we are already making an approximation by saying that all delays are exponentially distributed, this is a reasonable thing to do. Note that if we initially invoke the method $O.f$, we may want to set $R_E(O) = 0$ to mean that there is no communication delay between the user and the object O (i.e. the object O corresponds to the user's personal computer).

In this interpretation, we have collapsed internal and external operations into a single transition. When we want to model systems with multiple invocations, however, it is not always the case that the external operation can take place immediately — a server might be engaged with another client, for example. In the next chapter, we will show how to resolve this by mapping onto a PEPA model — we can use the semantics of cooperation in PEPA to ensure that the client blocks until the server becomes available.

3.5 Collecting Semantics of Siril

Recall that the ultimate aim of our semantics is to generate a performance model of a Siril program. In the continuous time interpretation that we have just seen, we associate *rates* with various operations so that we generate a CTMC. This is guaranteed to be finite, since there are no loops or recursive method calls in Siril. However, the state space might be very large, and more importantly, we lose the thread-level compositionality of the original Siril program. We can regain this however, to generate a *compositional* performance model, by means of a collecting semantics over the probabilistic interpretation.

One way to approach this is to consider a single method $O_i.f_j$ — this could be invoked from several places in the interpretation, corresponding to different paths in the probabilistic automaton of the caller. If we consider a stage $s \in [\![O_i.f_j]\!]^S_{pa}$, there might be multiple states of the form $\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{M,n_M}$, such that $s_{i,j} = s$, which correspond to being in this particular stage of $O_i.f_j$. If we just want to describe the behaviour of $O_i.f_j$, we could "project" each state in the interpretation, so that it only refers to its local stages. In other words, we can define an operator $\pi_{O_i.f_j}$ such that:

$$\pi_{O_i.f_j}(\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{M,n_M}) = \mu \vdash s_{i,j}$$

This would then induce a projected transition system. The problem, however, is that it still includes transitions between the same stage that correspond to a *different* method executing. As an example, we might have the following sequence of transitions:

$$\mu \vdash s \rightarrow \mu' \vdash s \rightarrow \mu'' \vdash s'$$

The measure μ is altered during the first transition by an external method because it contains information about *all* the variables in the system, and not just those that occur in $O_i.f_j$. To avoid this, we could try to project the measures themselves, so that they only describe the variables in $O_i.f_j$. The above transitions would then become as follows:

$$\pi_{O_i.f_j}(\mu) \vdash s \rightarrow \pi_{O_i.f_j}(\mu') \vdash s \rightarrow \pi_{O_i.f_j}(\mu'') \vdash s'$$

Since a method only modifies its local variables, $\pi_{O_i.f_j}(\mu) = \pi_{O_i.f_j}(\mu')$ — meaning that we introduce a non-deterministic choice between the transitions out of the state $\pi_{O_i.f_j}(\mu) \vdash s$. This is due to us throwing away information about the environment — namely, the other methods — when we project onto a particular method of the system. We could resolve this non-determinism by allowing the projected interpretations to be *driven* by one another. This is an idea that we will follow in detail in the next chapter, and more specifically in Section 4.5, when we show how to construct a PEPA model from the probabilistic interpretation of a Siril program.

The reader may at this point be wondering why we do not define such a collecting semantics here. The reason is that our probabilistic interpretation assumes that we have the ability to represent and manipulate arbitrary measures. In other words, it is not computable in general, so it is meaningless to define a collecting semantics on top of it. Even if we start with an initial measure that is compactly representable, the operations of the program will in general lead to one that is not. In the next chapter, we will solve this problem by building an *abstraction* of the semantics that over-approximates the behaviour of a program — but with the benefit that the analysis is computable, and the measures are efficiently representable. As a consequence, it will then be possible to extract a PEPA model from a Siril program.

Chapter 4

Stochastic Abstraction of Programs

In the previous chapter, we introduced the Simple Imperative Remote Invocation Language (SIRIL), and described its semantics in terms of probabilistic automata. The interpretation of this semantics results in a finite acyclic Markov chain, but this is not a feasible approach to generating a performance model in practice, for reasons that we will outline below. In this chapter, we will develop an *abstract interpretation* of SIRIL, which *will* allow us to feasibly generate performance models from SIRIL programs.

There are two main problems with our concrete semantics and probabilistic interpretation, for which we need to take additional steps to address:

1. The concrete semantics allows *general* measures over the state of a program's variables. There are two issues with this. Firstly, we need to be able to compactly describe a measure, if we want to efficiently execute the probabilistic interpretation — this is certainly not the case for measures in general. Secondly, when we operate on a measure (i.e. following the semantics of a SIRIL program), we also need to describe the resulting measure compactly[1]. We address these issues by restricting the possible measures to a particular form — the *truncated multivariate normal* measures — which can be represented and operated on efficiently. This leads to an *abstract interpretation* that over-approximates the concrete probabilistic interpretation. Crucially, the measures in the abstract interpretation will *not* in general be probability measures, since they over-approximate the probabilities in the concrete interpretation.

[1] As an example of how this might not be the case, consider an arithmetic expression $X_1 + X_2$, where X_1 is normally distributed, and X_2 is uniformly distributed over some interval. We can compactly represent the probability measures for both X_1 and X_2, but there is no compact representation for the sum of these measures.

2. The interpretation (both concrete and abstract) of a Siril program is not *compositional*. This means that we generate a single large Markov chain describing the whole system, which restricts the types of analysis we can perform. To combat this, we will introduce a *collecting semantics* that re-introduces thread-level compositionality, allowing us to construct a *PEPA model* of the system. From this, a wider range of tool support is available, and model-level transformations are easier.

The important point about the approach that we will describe in this chapter is that it is *safe* — any trace is at least as likely to happen in the abstract system as in the concrete system (we will formally define what we mean by this in Section 4.1). In addressing the first point, we can use our abstract interpretation to obtain an upper bound on the actual probability of a particular trace, or execution, of the program. For the second point, the collecting semantics allows us to construct a compositional performance model, without changing the underlying Markov chain that is described.

Note that there is an additional approximation inherent when we use a continuous time Markovian interpretation. By associating rates to the internal and external operations of a method, we are assuming that the time taken to perform each operation is exponentially distributed (and therefore history-independent). This approximation is necessary to make our performance analysis feasible, although the granularity at which we specify the rates is a design decision. With only minor modifications to our technique we might instead, for example, associate a different internal rate to each transition in a method's probabilistic automaton.

In this chapter, we will begin by introducing abstract interpretation as a framework for program analysis in Section 4.1. In Section 4.2, we show how this can be applied to Siril programs, by developing an abstract domain for probability measures based on the truncated multivariate normal distribution. Bringing this all together, we present an abstract semantics, abstract interpretation, and collecting semantics for Siril in Sections 4.3, 4.4 and 4.5 respectively. This follows the same pattern as the concrete case, except that we construct a compositional PEPA model of the program, rather than a Markov chain. We use the client-server example from the previous chapter as a running example.

An overview of this analysis framework is shown in Figure 4.1. Note that we have a safety relation between the concrete and abstract interpretations, which ensures the correctness of our analysis. Just like the concrete semantics of the previous chapter, there are three stages to deriving an *abstract* performance model of a program:

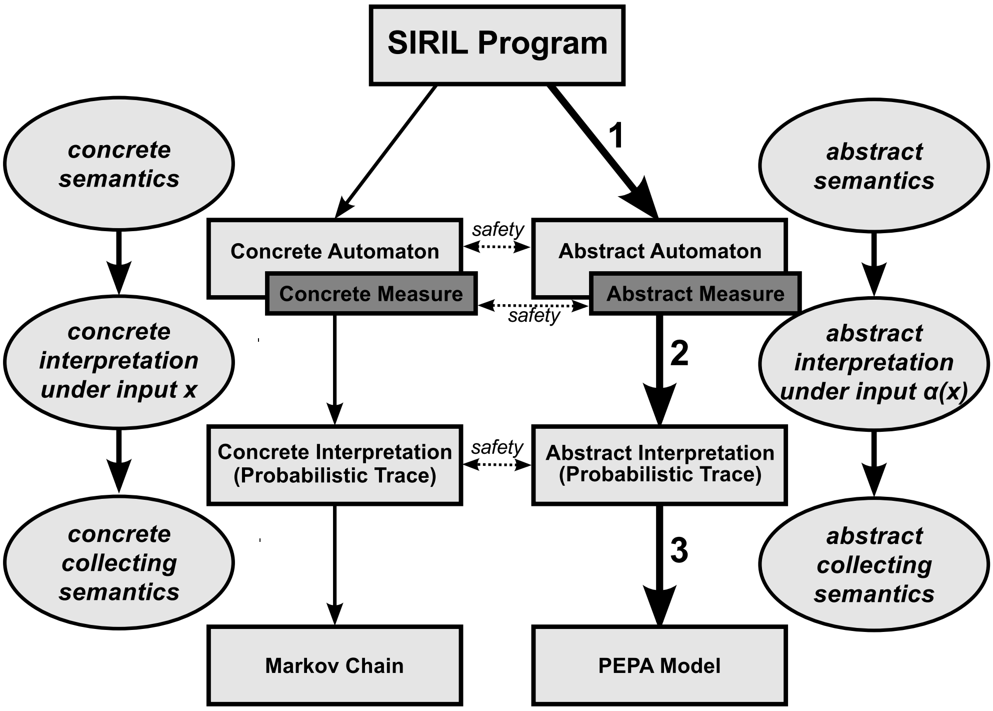

SIRIL Program
1
concrete semantics
abstract semantics
Concrete Automaton
safety
Abstract Automaton
Concrete Measure
safety
Abstract Measure
concrete interpretation under input x
2
abstract interpretation under input α(x)
Concrete Interpretation (Probabilistic Trace)
safety
Abstract Interpretation (Probabilistic Trace)
concrete collecting semantics
3
abstract collecting semantics
Markov Chain
PEPA Model

Figure 4.1: Overview of our probabilistic abstract interpretation

1. *Abstract semantics* [Section 4.3] — we compute an abstract probabilistic automaton that is a safe approximation of the concrete semantics of the program. In other words, we satisfy the relational homomorphism property (Definition 4.1.1).

2. *Abstract interpretation* [Section 4.4] — we "execute" the abstract automaton with a particular input measure. This generates a labelled transition system (which can be thought of as a set of traces) that safely approximates the concrete one, in that the measure associated with each state bounds that of the corresponding state in the concrete interpretation.

3. *Abstract collecting semantics* [Section 4.5] — for each method in the Siril program, we project the transitions in the abstract interpretation onto its local state space. By labelling these projected transitions with input and output measures (corresponding to receiving and sending arguments and return values), we can map onto a PEPA component describing the method — where input and output measures map onto action types. In Section 4.6, we will additionally consider model-level constructions of more complex instantiations — for example, having multiple methods that are initially invoked, or repeatedly invoking a method so that we can reason about steady state behaviour.

After developing this framework, we will then, in Section 4.6, look at some model-level transformations, such as instantiating a program with multiple clients initiating requests in parallel. Finally, we conclude in Section 4.7 with a discussion of how to modify our approach to produce performance models with non-determinism. This provides a link to the ideas of abstracting Markov chains in Chapter 5.

4.1 Program Analysis and Abstract Interpretation

Of the many techniques available for formally analysing programs, four of the main approaches are data flow analysis, control flow analysis, abstract interpretation, and type and effect systems [133]. Whilst all of these are widely used to analyse qualitative properties of programs, there have been relatively few applications to *probabilistic* and *stochastic* analyses. Most of the work that has been done in this regard involves probabilistic extensions to abstract interpretation [128, 141].

If we consider our goal of deriving a performance model from program code, we need to develop a suitable program analysis technique. More specifically, we need a program analysis for Siril programs, whose output is a *probabilistic abstraction* of the program[2]. To do this, we need to gather information about the possible *values* of the variables at each program point (i.e. a value of the program counter), so that we can reason about the probability of taking each control flow branch. The most natural choices of techniques to use are are *data flow analysis*, in which we gather a set of possible values for each node in the program's control flow graph, and *abstract interpretation*, in which we execute the program using an abstract domain of values.

These two techniques essentially do the same thing, but from different perspectives. In data flow analysis, the program is viewed as a static object (usually graph-structured), on which we annotate information. Abstract interpretation, on the other hand, views the program as a dynamic entity that is executed, and then outputs the desired information. In both cases, this 'information' is represented in a partially ordered abstract domain that exhibits two properties:

1. Every operation in the abstract domain *safely approximates* its counterpart in the concrete domain. This means that any value we obtain from a sequence of operations in the abstract domain over-approximates the actual, concrete value.

[2] The concrete semantics of Section 3.3 is probabilistic, and we only move to the stochastic domain by using the continuous time Markovian interpretation (described in Section 3.4.2), which adds timing information.

2. Any infinite sequence of operations on an element in the abstract domain will *converge*. This is achieved by ensuring two conditions — that the operations are *monotone* (i.e. they lead to ever increasing values in the abstract domain), and that the abstract domain has a *finite height* (i.e. there are no infinite ascending chains in the partially ordered set). This ensures termination of the analysis.

In the case of Siril, however, only the first of these properties is essential, since termination of the analysis is guaranteed in the absence of loops (see Appendix A for a discussion of how to deal with looping behaviour in Siril).

We choose to use abstract interpretation as our analysis framework because it is notationally more general than data flow analysis, which has historically been used only for particular program analyses. There is no reason why we cannot formulate our approach using data flow analysis, but we believe that this would lead to a less natural presentation[3]. In the remainder of this chapter, we will therefore focus our attention on abstract interpretation.

Classical abstract interpretation [50] is a mathematical framework that relates a concrete domain to an abstract one. Properties in the abstract domain are *safe approximations* (supersets) of their concrete counterparts. By constructing a suitable abstract domain and abstract semantics, we can reason about properties of a program that are in general undecidable — for example, the possible values that a variable can take throughout the program's execution — by not being precise in all cases.

Abstract interpretation can be applied to many different semantic frameworks, but since we are working with a transition system style of semantics, we will describe it just in this setting. Consider two partially ordered sets, a concrete domain $(X, \leq)$ and an abstract domain $(X^\sharp, \leq^\sharp)$. To relate these two domains, we have an *abstraction function*, $\alpha : X \to X^\sharp$, and a *concretisation function*, $\gamma : X^\sharp \to X$. A *safe abstraction* will be one that satisfies, for all $x \in X$, $x \leq \gamma(\alpha(x))$. If $X^\sharp \subset X$, we can let the concretisation function be the identity map so that we can concentrate solely on the definition of α.

The usefulness of this framework comes when we apply it to our transition semantics. If a state $x \in X$ (i.e. a valuation of variables) holds at a stage s (i.e. a program point), then we can write the statement $x \vdash s$. The concrete semantics induces a transition relation $\to$ between such statements. We then say that an abstract semantics, inducing $\to^\sharp$ for $x^\sharp \in X^\sharp$, is safe if the following holds [157]:

[3] Note that the choice between data flow analysis and abstract interpretation has no impact on our inability to deal with loops.

Definition 4.1.1. *A concrete and an abstract transition relation, $\rightarrow$ and $\rightarrow^{\sharp}$, satisfy the* relational homomorphism *property if $x_1 \vdash s_1 \rightarrow x_2 \vdash s_2$ and $\alpha(x_1) \leq^{\sharp} x_1^{\sharp}$ imply that there is an abstract transition $x_1^{\sharp} \vdash s_1 \rightarrow^{\sharp} x_2^{\sharp} \vdash s_2$ such that $\alpha(x_2) \leq^{\sharp} x_2^{\sharp}$.*

Intuitively, this means that if we start with an over-approximation of a property at an initial stage in the automaton, then the property at every stage in the abstract interpretation is guaranteed to over-approximate that of the corresponding stage in the concrete probabilistic interpretation. This is also known as a *subject reduction* result.

A very common way of constructing an abstract interpretation is to find a pair of monotone functions α and γ that form a *Galois connection* [50]. To do this, we need a notion of 'best' approximation, which does not always exist, and indeed does not exist for our domains[4]. This has the advantage of telling us *how* to construct our abstract semantics, as opposed to constructing it first and then proving that it is safe.

In the probabilistic setting, our domains are Banach spaces rather than preordered sets, but abstract interpretation can still be applied. This was first done by Monniaux [128], where measures are ordered by their total weight. In this way, distributions can be piece-wise compared by discretising them into point measures (for an intuition, think of the midpoint rule for numerical integration). Since in our case, we will be considering a particular class of measures, however, and we want a finer-grained distinction between them, we will use a stronger comparison. We call this the *strict ordering* on measures:

Definition 4.1.2. *Two measures μ and μ' over the same measurable space (X, σ_X) are comparable under the* strict ordering *on measures, denoted $\mu \leq_{\mathrm{str}} \mu'$, if:*

$$\forall x \in \sigma_X.\ \mu(x) \leq \mu'(x)$$

Our motivation for this ordering is that it allows the measures on any set to be compared. For example, we can compare the probability of taking a control-flow decision by looking at the measure on the set of values that satisfy the condition. Note that any measure that over-approximates a probability measure (other than itself) will be a super-probability measure — whatever probability the abstraction gives, we will know that the actual probability is less than or equal to this, and can never be greater.

An alternative approach taken by Di Pierro, Wiklicky *et al.* [141, 140] is to look for a probabilistic analogue of the Galois connection. This, the Moore-Penrose pseudo

[4] Intuitively, this is because we use normal (Gaussian) measures in our abstract domain, as we shall see in Section 4.2. Since we can vary both the mean and variance, we cannot construct a unique 'best' abstraction for a given measure.

inverse, gives the closest approximation to the inverse of a function, leading to a *probabilistic* notion of safety. While this approach has had much success, it is difficult to use in practice for infinite Banach spaces (i.e. continuous measures) such as the ones we consider, as we would need to construct a finite representation of the abstraction and concretisation functions.

4.2 Abstraction of Siril Programs

In the concrete semantics of Siril that we presented in Section 3.3, we allowed any probability measure over the state of a program's variables. In a practical sense, however, it is infeasible to deal with general distributions, since we need to be able to represent them somehow. Rather than taking the approach by Monniaux [128] — of discretising the distributions — we will look for a suitable class of *continuous* distributions that are easily parameterised, and can be efficiently manipulated in the abstract domain.

Such a class of distributions are the *multivariate normal distributions* [176], and more generally the *truncated multivariate normal distributions*. The appeal of normal distributions is that they are closed under linear operations (i.e. addition, subtraction, and multiplication by a constant), and are commonly observed in practice, due to the central limit theorem. By using a multivariate distribution, we can record the dependencies between variables in a compact way. Truncations can be used to represent control-flow constraints, which restrict the range of values that the variables can take.

It is important to remember that these are *measures*, and not *distributions* in general. Since we over-approximate the measures in the concrete domain, it is possible for the total weight to be greater than one, and when we truncate a measure, we eliminate part of the probability mass so that its total weight is less than one. It is always possible to normalise a non-zero, finite measure μ to a probability measure by multiplying it by a constant factor $\frac{1}{\mu_T}$, where μ_T is the total weight of μ. If μ is a measure over (X, σ_X), this means that there is a probability measure μ' over (X, σ_X) such that for all $Y \in \sigma_X$, $\mu(Y) = \mu_T \mu'(Y)$.

A more convenient way of describing many measures, is to do so in terms of a *density function*. For a measure μ over (X, σ_X), we say that it has a density function $f : X \to \mathbb{R}_{\geq 0}$ if for all $Y \in \sigma_X$:

$$\mu(Y) = \int_Y f \, dm^* = \int_Y f(y) \, dm^*(y)$$

which is the Lebesgue integral of f over the measurable set Y, with respect to the Lebesgue measure m^* [42] (see Section 3.2). In the following, we define a measure in terms of its density function, using the above equation:

Definition 4.2.1. *A* multivariate normal measure*, μ, which we write as $\mu_T N_N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, is a measure over $(\mathbb{R}^N, \mathcal{M}^{(N)})$ with the following density function (for $\boldsymbol{x} \in \mathbb{R}^N$):*

$$f(\boldsymbol{x}) = \frac{\mu_T}{|\boldsymbol{\Sigma}|^{\frac{1}{2}}(2\pi)^{\frac{N}{2}}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}$$

where N is the number of variables, $\boldsymbol{\mu}$ is the mean vector of length N, and $\boldsymbol{\Sigma}$ is the $N \times N$ covariance matrix. Note that $\boldsymbol{M}^{\mathrm{T}}$ denotes the matrix transpose of $\boldsymbol{M}$. Considering $\frac{\mu}{\mu_T}$ as the joint distribution of values of a vector $\boldsymbol{X}$ of N variables, the elements of $\boldsymbol{\Sigma}$ are such that $\boldsymbol{\Sigma}(i,i) = \mathrm{Var}(\boldsymbol{X}(i))$, and $\boldsymbol{\Sigma}(i,j) = \mathrm{Cov}(\boldsymbol{X}(i), \boldsymbol{X}(j)) = \boldsymbol{\Sigma}(j,i)$.

It is unfortunate that the Greek letter μ is conventionally used for both measures and means, however since we are dealing with multivariate distributions, we will from hereon always use boldface $\boldsymbol{\mu}$ to refer to the mean, and lightface μ for measures.

In order to allow *truncated* multivariate normal measures, we define a truncation function $T[\boldsymbol{a}, \boldsymbol{b}]$ over measures:

Definition 4.2.2. *The truncation function $T[\boldsymbol{a}, \boldsymbol{b}]$, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are column vectors of length N, is defined over measures μ over $(\mathbb{R}^N, \mathcal{M}^{(N)})$, such that (for $X \in \mathcal{M}^{(N)}$):*

$$T[\boldsymbol{a}, \boldsymbol{b}](\mu)(X) = \mu(X \cap \{\boldsymbol{x} \in \mathbb{R}^N \mid \boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b}\})$$

The elements of $\boldsymbol{a}$ and $\boldsymbol{b}$ are from the set $\mathbb{R} \cup \{\bot, \top\}$, where $\forall x \in \mathbb{R}.\ \bot < x < \top$. The ordering on column vectors is such that $\boldsymbol{x} \leq \boldsymbol{y}$ iff $\boldsymbol{x}_i \leq \boldsymbol{y}_i$ for all $1 \leq i \leq n$.

Intuitively, the truncation $T[\boldsymbol{a}, \boldsymbol{b}]$ confines measures to the rectangular region $[\boldsymbol{a}, \boldsymbol{b}]$, such that the measure of any set outside this region is zero. We can now define the class of truncated multivariate normal measures:

Definition 4.2.3. *A measure μ is a* truncated multivariate normal measure *if it can be written in the form $T[\boldsymbol{a}, \boldsymbol{b}]\mu_T N_N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. If $\mu_T = 1$ — i.e. we had a probability measure before performing the truncation — then we will write $\mu = T[\boldsymbol{a}, \boldsymbol{b}]N_N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.*

Before we describe our abstraction function α and our abstract semantics, let us recall an important property of the multivariate normal distribution. For a multivariate normally distributed vector of random variables $\boldsymbol{X} \sim N_N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, if we apply a linear operation $\lambda \boldsymbol{X}.\ \boldsymbol{BX}+\boldsymbol{c}$, where $\boldsymbol{B}$ is an $N\times N$ matrix and $\boldsymbol{c}$ is a column vector of size N, the following standard result [176] holds:

$$\boldsymbol{Y} = \boldsymbol{BX}+\boldsymbol{c} \sim N_N(\boldsymbol{B\mu}+\boldsymbol{c}, \boldsymbol{B\Sigma B}^T)$$

So that we can directly talk about an operation on measures, rather than on random variables, we introduce the operator $L[\boldsymbol{B},\boldsymbol{c}]$, which is defined as follows:

Definition 4.2.4. *The linear operator function* $L[\boldsymbol{B},\boldsymbol{c}]$*, where* $\boldsymbol{B}$ *is an* $N\times N$ *matrix and* $\boldsymbol{c}$ *is a column vector of length N, is defined over measures* μ *over* $(\mathbb{R}^N, \mathcal{M}^{(N)})$*, such that (for* $X\in\mathcal{M}^{(N)}$*):*

$$L[\boldsymbol{B},\boldsymbol{c}](\mu)(X) = \mu(\{\boldsymbol{x} \mid \boldsymbol{Bx}+\boldsymbol{c}\in X\})$$

A consequence from the standard properties of multivariate normal distributions is that if $\mu = \mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})$ *then* $L[\boldsymbol{B},\boldsymbol{c}](\mu) = \mu_T N_N(\boldsymbol{B\mu}+\boldsymbol{c}, \boldsymbol{B\Sigma B}^T)$*.*

Note that our use of the term ‘linear’ in this context is more specific than the usual definition of a linear operator over measures — namely that an operator M is linear if, for all $c\in\mathbb{R}$, $M(c\mu) = cM(\mu)$ and $M(\mu_1+\mu_2) = M(\mu_1)+M(\mu_2)$. Our operator *is* linear in this sense, but its name comes from the fact that it represents a linear update to the program's variables. A consequence is that $L[\boldsymbol{B},\boldsymbol{c}]$ is monotonic: if $\mu_1 \leq_{\text{str}} \mu_2$ then $L[\boldsymbol{B},\boldsymbol{c}](\mu_1) \leq_{\text{str}} L[\boldsymbol{B},\boldsymbol{c}](\mu_2)$.

Although we can easily apply linear operations precisely to multivariate normal measures, the same is not true of truncated multivariate normal measures — the class of truncated multivariate normal measures is not closed under such operations [98]. To combat this, we will introduce an *abstract* linear operator function $L^\sharp[\boldsymbol{B},\boldsymbol{c}]$:

Definition 4.2.5. *The abstract linear operator function* $L^\sharp[\boldsymbol{B},\boldsymbol{c}]$ *is defined over truncated multivariate normal measures* $T[\boldsymbol{a},\boldsymbol{b}]\mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})$*, such that:*

$$L^\sharp[\boldsymbol{B},\boldsymbol{c}](T[\boldsymbol{a},\boldsymbol{b}]\mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})) = T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c})\mu_T N_N(\boldsymbol{B\mu}+\boldsymbol{c}, \boldsymbol{B\Sigma B}^T)$$

where $\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}$ *is defined as per interval analysis*[5] *[129].*

The safety of this abstraction is established in the following theorem, which allows us to over-approximate the measure that we would get from applying the concrete linear operator function to a truncated multivariate normal measure, by applying the linear operator *before* the truncation.

[5] For an interval $[a,b] = \{x\in\mathbb{R} \mid a\leq x\leq b\}$, we define addition as $[a,b]+[c,d] = [a+c, b+d]$, and multiplication by a constant as $c[a,b] = [ca,cb]$ if $c\geq 0$ and $[cb,ca]$ otherwise.

Theorem 4.2.6. *For all measures μ, $L[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu) \leq_{\mathrm{str}} L^\sharp[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)$.*

Proof: By definition of the operators, and since we can safely apply the new truncation interval $T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c})$ first (values outside this region are impossible to obtain), we have:

$$\begin{aligned}
L[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)(X) &= \\
& T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)(X) \\
=\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu(X \cap \{\boldsymbol{x} \in \mathbb{R}^N \mid \boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b}\})) \\
=\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ \mu(\{\boldsymbol{x} \mid \boldsymbol{B}\boldsymbol{x}+\boldsymbol{c} \in X \cap \{\boldsymbol{x} \in \mathbb{R}^N \mid \boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b}\}\}) \\
\leq_{\mathrm{str}}\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ \mu(\{\boldsymbol{x} \mid \boldsymbol{B}\boldsymbol{x}+\boldsymbol{c} \in X\}) \\
=\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu)(X) = L^\sharp[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)(X)
\end{aligned}$$

■

4.2.1 Relating the Concrete and Abstract Domains

We would like our concrete domain to consist of all possible measures, and our abstract domain to be the truncated multivariate normal measures, as described. Unfortunately, constructing an abstraction function from such a domain is not a simple task. Not only does it contain measures that we cannot write down, but it is difficult to satisfy the relational homomorphism property (Definition 4.1.1). Instead we restrict our concrete domain to those measures that can be computed by a series of linear operations and truncations applied to a multivariate normal measure. Whilst this is restrictive, it still allows us to represent a useful class of measures, and we hope to relax this requirement in the future. More formally:

- Our *concrete domain* $\mathcal{D}$ consists of measures of the form $L[\boldsymbol{B}_n,\boldsymbol{c}_n] \circ T[\boldsymbol{a}_n,\boldsymbol{b}_n] \circ \ldots \circ L[\boldsymbol{B}_1,\boldsymbol{c}_1] \circ T[\boldsymbol{a}_1,\boldsymbol{b}_1]\mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})$, and is ordered by $\leq_{\mathrm{str}}$.
- Our *abstract domain* $\mathcal{D}^\sharp$ consists of measures of the form $T[\boldsymbol{a},\boldsymbol{b}]\mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})$, and is also ordered by $\leq_{\mathrm{str}}$.

Note that $\mathcal{D}^\sharp \subset \mathcal{D}$, and that $\mathcal{D}^\sharp$ is just the set of truncated multivariate normal measures. We can now define our abstraction function as follows:

Definition 4.2.7. *The abstraction function $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ of a measure $\mu \in \mathcal{D}$ is defined inductively as follows:*

$$\begin{aligned}
\alpha(\mu) &= \mu \quad \textit{if } \mu \textit{ is a multivariate normal measure} \\
\alpha(T[\boldsymbol{a},\boldsymbol{b}](\mu)) &= T[\boldsymbol{a},\boldsymbol{b}](\alpha(\mu)) \\
\alpha(L[\boldsymbol{B},\boldsymbol{c}](\mu)) &= L^\sharp[\boldsymbol{B},\boldsymbol{c}](\alpha(\mu))
\end{aligned}$$

Note that $T[\boldsymbol{a}_2,\boldsymbol{b}_2] \circ T[\boldsymbol{a}_1,\boldsymbol{b}_1] = T([\boldsymbol{a}_2,\boldsymbol{b}_2] \cap [\boldsymbol{a}_1,\boldsymbol{b}_1])$ if the intersection of the intervals is non-empty, and $\lambda x.0$ (the zero measure) otherwise. We can also compose linear operators, since $L[\boldsymbol{B}_2,\boldsymbol{c}_2] \circ L[\boldsymbol{B}_1,\boldsymbol{c}_1] = L[\boldsymbol{B}_2\boldsymbol{B}_1,\boldsymbol{B}_2\boldsymbol{c}_1 + \boldsymbol{c}_2]$.

Our concretisation function, $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$, is simply the identity map — $\gamma(\mu) = \mu$ for all $\mu \in \mathcal{D}^\sharp$. We state and prove the safety of this abstraction as follows:

Theorem 4.2.8. *The abstraction specified by α and γ is safe — namely, for all measures $\mu \in \mathcal{D}$, $\mu \leq_{\text{str}} \gamma(\alpha(\mu))$.*

Proof: Since γ is the identity map, we need to prove that $\mu \leq_{\text{str}} \alpha(\mu)$. We prove this by induction on the structure of μ, which is of the form $M_n \circ \ldots \circ M_1 \mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})$, for some $n \geq 0$. In the base case, when $n = 0$, $\mu = \mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})$, hence $\alpha(\mu) = \mu$, since μ is a multivariate normal measure, and so $\mu \leq_{\text{str}} \alpha(\mu)$.

In the inductive case, we assume that for $\mu = M_n \circ \ldots \circ M_1 \mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})$, $\mu \leq_{\text{str}} \alpha(\mu)$. To show that $M_{n+1}(\mu) \leq_{\text{str}} \alpha(M_{n+1}(\mu))$, we need to consider the two cases of M_{n+1}:

1. If $M_{n+1} = T[\boldsymbol{a},\boldsymbol{b}]$, then we have $\alpha(T[\boldsymbol{a},\boldsymbol{b}](\mu)) = T[\boldsymbol{a},\boldsymbol{b}](\alpha(\mu))$. But from the induction hypothesis, $\mu \leq_{\text{str}} \alpha(\mu)$. Hence $T[\boldsymbol{a},\boldsymbol{b}](\mu) \leq_{\text{str}} T[\boldsymbol{a},\boldsymbol{b}](\alpha(\mu)) = \alpha(T[\boldsymbol{a},\boldsymbol{b}](\mu))$, since $T[\boldsymbol{a},\boldsymbol{b}]$ is monotone.

2. If $M_{n+1} = L[\boldsymbol{B},\boldsymbol{c}]$, then we have $\alpha(L[\boldsymbol{b},\boldsymbol{c}](\mu)) = L^\sharp[\boldsymbol{B},\boldsymbol{c}](\alpha(\mu))$. But from the induction hypothesis, $\mu \leq_{\text{str}} \alpha(\mu)$. Hence $L[\boldsymbol{B},\boldsymbol{c}](\mu) \leq_{\text{str}} L^\sharp[\boldsymbol{B},\boldsymbol{c}](\mu) \leq_{\text{str}} L^\sharp[\boldsymbol{B},\boldsymbol{c}](\alpha(\mu)) = \alpha(L[\boldsymbol{B},\boldsymbol{c}](\mu))$, as a consequence of Theorem 4.2.6 and the fact that $L[\boldsymbol{B},\boldsymbol{c}]$ is monotone.

■

4.3 Abstract Semantics of Siril

The abstract semantics of each method in a Siril program is an automaton with the same stages as its concrete semantics, but with transitions that operate on truncated multivariate normal measures rather than arbitrary measures. Since the structure of the automaton is the same as for the concrete semantics, we need only present an abstraction for the measure operators on the transitions. We therefore need to define abstract operators for assignment, $[\![\cdot]\!]^\sharp_p$, and for the branching operators, $e^\sharp_{[\![\cdot]\!]}$ and $e^\sharp_q$ (for $q \in [0,1]$). Once we have done so, the abstract probabilistic automaton semantics of a Siril method $O.f$ is given by:

$$[\![O.f]\!]^\sharp_{pa} = [\![O.f]\!]_{pa}\{\, [\![\cdot]\!]^\sharp_p \;/\; [\![\cdot]\!]_p,\; e^\sharp_{[\![\cdot]\!]} \;/\; e_{[\![\cdot]\!]},\; e^\sharp_q \;/\; e_q \}$$

Let us start with the abstract operator for assignment, $[\![\cdot]\!]^\sharp_p$:

$$[\![O.f \mid X_i := E]\!]^\sharp_p(\mu) = L^\sharp[\boldsymbol{B},\boldsymbol{c}](\mu)$$

where $\boldsymbol{B}$ and $\boldsymbol{c}$ describe the operation of E, such that $[\![X_i{:=}E]\!](\boldsymbol{x}) = \boldsymbol{Bx}+\boldsymbol{c}$, for a state $\boldsymbol{x}$ of the program's variables (a column vector of length N). This is possible because of the restriction in the Siril language to *linear* arithmetic expressions, as motivated in Section 3.1. Rather than applying the linear operator $L[\boldsymbol{B},\boldsymbol{c}]$, we use the abstract operator from Definition 4.2.5, which over-approximates the actual measure by applying the linear update *before* any truncation operators.

Note that by this definition, the abstract semantics is not compositional — as in the concrete case, it assumes a common set of variables over the entire system, hence remote procedure calls only make sense when the method we call is present as part of the system. This will also be the case for our abstract interpretation, as we shall see in Section 4.4, and it is only when we present our collecting semantics in Section 4.5 that we will re-introduce compositionality.

For the abstract semantics of conditional operators, we have the following:

$$\begin{array}{lcllcl}
e^\sharp_{[\![\mathtt{true}]\!]}(\mu) & = & \mu & e^\sharp_{[\![\neg\mathtt{true}]\!]}(\mu) & = & \lambda x.\,0 \\
e^\sharp_{[\![X_i \le c]\!]}(\mu) & = & T[\bot,\boldsymbol{a}](\mu) & e^\sharp_{[\![\neg(X_i < c)]\!]}(\mu) & = & T[\boldsymbol{b},\top](\mu) \\
e^\sharp_{[\![X_i < c]\!]}(\mu) & = & T[\bot,\boldsymbol{b}](\mu) & e^\sharp_{[\![\neg(X_i \le c)]\!]}(\mu) & = & T[\boldsymbol{a},\top](\mu) \\
\multicolumn{6}{l}{e^\sharp_q(T[\boldsymbol{a},\boldsymbol{b}]\mu_T N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})) = T[\boldsymbol{a},\boldsymbol{b}](q.\mu_T)N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})}
\end{array}$$

where $\boldsymbol{a}_i = c+0.5$, $\boldsymbol{a}_j = \top$ (for $j \neq i$), and $\boldsymbol{b}_i = c-0.5$, $\boldsymbol{b}_j = \bot$ (for $j \neq i$). We need to add or subtract 0.5, because the variables in our language have *discrete, integer* values, whereas our semantics is in terms of *continuous* measures. Note that there is a slight over-approximation for the $\{<,>\}$ comparisons, to avoid distinguishing between open and closed truncation intervals in the abstract domain. The probabilistic branching operator has the same semantics as in the concrete case.

It remains to prove that the abstract semantics is safe; that is to say, that it satisfies the relational homomorphism property (Definition 4.1.1).

Theorem 4.3.1. *Consider a* Siril *program P, with concrete semantics $[\![P]\!]_{pa}$ and abstract semantics $[\![P]\!]^\sharp_{\mathrm{pa}}$. For all transitions $s_1 \xrightarrow{M} s_2 \in [\![P]\!]_{pa}$ there exists an abstract transition $s_1 \xrightarrow{M^\sharp} s_2 \in [\![P]\!]^\sharp_{\mathrm{pa}}$ such that for all measures $\mu \in \mathcal{D}$, if $\alpha(\mu) \le_{\mathrm{str}} \mu^\sharp \in \mathcal{D}^\sharp$ then $\alpha(M(\mu)) \le_{\mathrm{str}} M^\sharp(\mu^\sharp)$.*

Proof: Firstly, we note that there is a bijection between concrete and abstract transitions, which ensures a unique $M^\sharp$ for each M. M and $M^\sharp$ both consist of a sequence of truncation and linear operators (ignoring the identity operator as trivial). We prove the theorem by induction on the length of this sequence, starting with the two base cases when the length is one.

For a truncation operator, we note that the abstract semantics generates an interval that over-approximates the actual set of values that satisfy the condition. Hence if $\alpha(M(\mu)) = \alpha(T[\boldsymbol{a},\boldsymbol{b}](\mu)) = T[\boldsymbol{a},\boldsymbol{b}](\alpha(\mu))$ (using the definition of α), then $M^\sharp(\mu^\sharp) = T[\boldsymbol{a}',\boldsymbol{b}'](\mu^\sharp)$, where $[\boldsymbol{a},\boldsymbol{b}] \subseteq [\boldsymbol{a}',\boldsymbol{b}']$. Hence $\alpha(M(\mu)) \leq_{\text{str}} M^\sharp(\mu^\sharp)$ since $\alpha(\mu) \leq_{\text{str}} \mu^\sharp$.

If M and $M^\sharp$ are linear operators then they have the forms $L[\boldsymbol{B},\boldsymbol{c}]$ and $L^\sharp[\boldsymbol{B},\boldsymbol{c}]$ respectively. Let $\alpha(\mu) = T[\boldsymbol{a}_1,\boldsymbol{b}_1](\mu_1)$ and $\mu^\sharp = T[\boldsymbol{a}_2,\boldsymbol{b}_2](\mu_2)$, such that $\alpha(\mu) \leq_{\text{str}} \mu^\sharp$. Then, using the monotonicity of $L[\boldsymbol{B},\boldsymbol{c}]$: if $\mu_1 \leq_{\text{str}} \mu_2$ then $L[\boldsymbol{B},\boldsymbol{c}](\mu_1) \leq_{\text{str}} L[\boldsymbol{B},\boldsymbol{c}](\mu_2)$:

$$\begin{aligned}
\alpha(L[\boldsymbol{B},\boldsymbol{c}](\mu)) &= L^\sharp[\boldsymbol{B},\boldsymbol{c}](\alpha(\mu)) \text{ from the definition of } \alpha \\
&= L^\sharp[\boldsymbol{B},\boldsymbol{c}](T[\boldsymbol{a}_1,\boldsymbol{b}_1](\mu_1)) \\
&= T(\boldsymbol{B}[\boldsymbol{a}_1,\boldsymbol{b}_1]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu_1) \\
&\leq_{\text{str}} T(\boldsymbol{B}[\boldsymbol{a}_2,\boldsymbol{b}_2]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu_2) \text{ since } \alpha(\mu) \leq_{\text{str}} \mu^\sharp \\
&= L^\sharp[\boldsymbol{B},\boldsymbol{c}](T[\boldsymbol{a}_2,\boldsymbol{b}_2](\mu_2)) = L^\sharp[\boldsymbol{B},\boldsymbol{c}](\mu^\sharp)
\end{aligned}$$

Finally, the inductive step completes the proof. Our induction hypothesis is that the sequence of operators M_n and $M_n^\sharp$ satisfy the condition that if $\alpha(\mu) \leq_{\text{str}} \mu^\sharp$ then $\alpha(M_n(\mu)) \leq_{\text{str}} M_n^\sharp(\mu^\sharp)$. Let $M_{n+1} = M \circ M_n$ and $M_{n+1}^\sharp = M^\sharp \circ M_n^\sharp$, such that M and $M^\sharp$ are base operators. Since M and $M^\sharp$ are either both truncation operators or both linear operators, it follows that $\alpha(M \circ M_n(\mu)) \leq_{\text{str}} M^\sharp \circ M_n^\sharp(\mu^\sharp)$ holds by the above cases. ■

4.4 Abstract Interpretation of Siril

As with the concrete semantics of the previous chapter, we can interpret the abstract semantics of a Siril program in the context of an instantiation of one of its methods, $O.f(X_1,\ldots,X_n)$. We will only consider an instantiation to be a single method call in this section, but we will show how to handle more complex instantiations in Section 4.6. The initial values of the variables $X_1,\ldots,X_n$ are given by an initial probability measure μ_I, which must be an element of our abstract domain — a truncated multivariate normal measure.

Let us first recall the structure of a Siril program, for the purposes of our abstract interpretation. This was given in Equation 3.3 in the previous chapter, which we repeat below:

$$O_1.f_1 \mid\mid \cdots \mid\mid O_1.f_{n_1} \mid\mid \cdots \mid\mid O_M.f_1 \mid\mid \cdots \mid\mid O_M.f_{n_M}$$

It is important to emphasise once more that this structure is purely for the purposes of carrying out the interpretation. When we come to construct a PEPA model from our abstract interpretation, we will ensure that only *one* method on an object can be instantiated at any one time, since an object corresponds to a single shared resource. A good way to think about it is that the abstract interpretation lets us compute the probability of each control-flow branch *if* the remote procedure calls are made. The fact that we might block for some time before the call proceeds is captured by the performance model — in our case, a PEPA model — that we construct from this.

We will first consider the abstract interpretation of a single method $O.f$, without remote procedure calls. Starting in an initial state $\mu_I^\sharp \vdash \circ_{O.f}$, where $\mu_I^\sharp$ is the initial measure and $\circ_{O.f}$ is the starting stage in the probabilistic automaton semantics of $O.f$, we can construct a labelled transition system in the same way as the concrete interpretation. The reachable states are all of the form $\mu^\sharp \vdash s$, where $s \in [\![O.f]\!]_{pa}^{S\sharp} = [\![O.f]\!]_{pa}^{S}$, and $\mu^\sharp \in \boldsymbol{B}(\mathbb{R}^N, \mathcal{M}^{(N)})$ is a measure over the state of the variables $X_1, \ldots, X_N$ in the method. The only difference from the concrete interpretation is that $\mu^\sharp$ will always be a truncated multivariate normal measure, given that the initial measure $\mu_I^\sharp$ is.

The abstract interpretation of $O.f$ is defined as follows — this is the same as the concrete interpretation in Equation 3.2, but over the abstract semantics $[\![\cdot]\!]_{pa}^{\sharp}$:

$$\mu^\sharp \vdash s \xrightarrow{p^\sharp}{}^{\sharp} \mu'^\sharp \vdash s' \;\text{ iff }\; s \xrightarrow{M^\sharp} s' \in [\![O.f]\!]_{pa}^{T\sharp} \wedge \mu'^\sharp = M^\sharp(\mu^\sharp)$$

This differs from the concrete interpretation, however, in the way we define the labels $p^\sharp$ on the transitions. If we were to use the same technique as the concrete interpretation — taking the ratio between the total weight after taking the transition ($\mu'^\sharp$) and that before ($\mu^\sharp$) — then the definition of $p^\sharp$ would be as follows:

$$p^\sharp = \frac{(M^\sharp(\mu^\sharp))(\mathbb{R}^N)}{\mu^\sharp(\mathbb{R}^N)} \tag{4.1}$$

In the concrete interpretation, this is the *probability* of taking the transition — in particular, the values of the labels on the out-going transitions of a state always sum to one. This is ensured by Theorem 3.4.1, which states that the total weight of the measure going into a stage is equal to that of the measure exiting the stage. In the abstract semantics, however, we *over-approximate* the measures, and so the sum of the values may be greater than one.

This leads to the question: what does this definition of $p^\sharp$ mean? We know that it is not a probability, but neither is it an *upper bound* of the corresponding probability in the concrete interpretation. To see this, notice that although the abstract numerator ($\mu'^\sharp(\mathbb{R}^N) = (M^\sharp(\mu^\sharp))(\mathbb{R}^N)$) is an upper bound of the concrete numerator, the abstract denominator ($\mu^\sharp(\mathbb{R}^N)$) is also an upper bound of the concrete denominator. Hence we cannot conclude that $p^\sharp$ is an upper-bounding probability.

Because of this, we cannot use the definition of $p^\sharp$ in Equation 4.1. Instead, we propose two alternative ways of labelling the abstract interpretation, depending on whether we want to build an approximate stochastic model, or a safe abstract model (containing non-determinism). We use the notation $p^\sharp(\mu^\sharp \vdash s \rightarrow \mu'^\sharp \vdash s')$ to denote the label $p^\sharp$ on the transition $\mu^\sharp \vdash s \rightarrow \mu'^\sharp \vdash s'$. Our two alternatives are:

1. Label the transitions with *approximate probabilities*, so that we construct a purely stochastic process. Since the abstract semantics may result in a greater total weight after exiting a stage than before entering it, we need to normalise with respect to the exit measure. This is given as follows:

$$p^\sharp(\mu^\sharp \vdash s \rightarrow \mu'^\sharp \vdash s') = \frac{\mu'^\sharp(\mathbb{R}^N)}{\displaystyle\sum_{\{\mu''^\sharp \mid \exists s''.\ \mu^\sharp \vdash s \rightarrow \mu''^\sharp \vdash s''\}} \mu''^\sharp(\mathbb{R}^N)} \tag{4.2}$$

   Note that in the concrete interpretation, this approach would give precisely the same probabilities as Equation 4.1, thanks to Theorem 3.4.1.

2. Label the transitions with upper and lower *bounding probabilities*. In other words, the label is an interval of probabilities, $[p^\sharp_L, p^\sharp_U]$, that contains the concrete transition probability p: $p^\sharp_L \leq p \leq p^\sharp_U$.

Of these two approaches, the first results in a (discrete time) Markov chain, whilst the second results in an abstract — or interval — Markov chain [63, 102]. This is a variant of a Markov Decision Process (MDP) [146], and contains non-determinism because there are many possible distributions that satisfy the constraints on the transition probabilities out of a state. Because this is not a purely stochastic model, and we wish to ultimately construct a PEPA model from our abstract interpretation, we will choose the first approach over the second — even though it is an approximation. This is not to say, however, that the second approach should be disregarded, and we discuss how we might go about adopting it in Section 4.7.

At this point, it is useful to observe that the abstract interpretation might result in highly unlikely states being considered. For example, if we begin with a variable $X \sim N(10, 1)$, then the probability that X is less than zero is negligibly small. To avoid constructing models with very small transition probabilities, which can lead to numerical problems, it is therefore useful to impose a numerical threshold on transition probabilities — if the probability is below some small value ϵ, then we round it to zero, and consider the next state unreachable. Whilst this is an approximation, it is sensible in the context of average behaviour, where we do not want to consider extremely rare events. As this is more of a practical concern, we will not formally include it in our abstract interpretation — it will come into play, however, when we consider an example.

We have now examined an abstract interpretation for individual Siril methods, but in order for this to be useful we need to deal with remote procedure calls. There are two ways in which we might extend our approach to handle this. The first, which we shall take, is to lift the abstract interpretation of methods to the entire system as was done in the concrete interpretation. In other words, we interpret a Siril program — which consists of many objects in parallel — as a whole. The second approach would be to carry out the abstract interpretation compositionally, and achieve a more compact representation at the expense of over-approximating the possible behaviours of the program. In our case, however, only one method can be executing at any time, and so we do not need the additional complexity of this approach. Instead, we will obtain compositionality through the collecting semantics, described in Section 4.5. Note that we will consider only the discrete time interpretation here, as we will introduce the rates as part of the collecting semantics.

The basic idea is precisely the same as that for the concrete probabilistic interpretation in the previous chapter. Given the abstract interpretation of individual methods, we can define an abstract interpretation over states of the form $\mu \vdash s_{1,1} \parallel \cdots \parallel s_{m,n_m}$, where for every method $O_i.f_j$ of every object, $s_{i,j}$ denotes the current stage in the method's probabilistic automaton semantics. This is described by the following, which are direct modifications of Equations 3.5 and 3.6, for abstract measures:

$$\mu^\sharp \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{m,n_m} \xrightarrow{p} \mu'^\sharp \vdash s_{1,1} \parallel \cdots \parallel \bullet_{O_i.f_j} \parallel \cdots \parallel s_{m,n_m}$$
$$\text{iff } \mu^\sharp \vdash s_{i,j} \xrightarrow{p} \mu'^\sharp \vdash \bullet_{O_i.f_j} \wedge \neg \mathit{blocked}(s_{i,j})$$

$$\mu^\sharp \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{i',j'} \parallel \cdots \parallel s_{m,n_m} \xrightarrow{p} \mu'^\sharp \vdash s_{1,1} \parallel \cdots \parallel s'_{i,j} \parallel \cdots \parallel \circ_{O_{i'}.f_{j'}} \parallel \cdots \parallel s_{m,n_m}$$
$$\text{iff } \mu^\sharp \vdash s_{i,j} \xrightarrow{p} \mu'^\sharp \vdash s'_{i,j} \wedge s'_{i,j} = s[O_{i'}.f_{j'}] \wedge \neg \mathit{blocked}(s_{i,j})$$

where $\mathit{blocked}(s_{i,j})$ is defined as in Equation 3.7.

If we instantiate a Siril program with a call to the method $O_i.f_j(X_1,\ldots,X_n)$, then the abstract interpretation is the probabilistic tree induced by applying the above transition rules to the initial state:

$$\mu_I^\sharp \vdash \bullet_{O_1.f_1} \parallel \cdots \parallel \bullet_{O_i.f_{j-1}} \parallel \circ_{O_i.f_j} \parallel \bullet_{O_i.f_{j+1}} \parallel \cdots \parallel \bullet_{O_n.f_{nm}}$$

This is the same as Equation 3.4, except that $\mu_I^\sharp$ must be a truncated multivariate normal distribution, rather than a general measure. Importantly, this describes the initial distribution of the variables in the Siril program, and we will get a *different* abstract interpretation for different values of $\mu_I^\sharp$. We require the user to specify this distribution, since this corresponds to analysing the program in a particular operating environment — for example, a web service application under a particular distribution of requests from clients. It would be cumbersome for the user to have to specify this distribution directly, however, since only the initial distribution of the argument variables should need to be specified — instead, we can characterise the abstract interpretation by a smaller set of parameters.

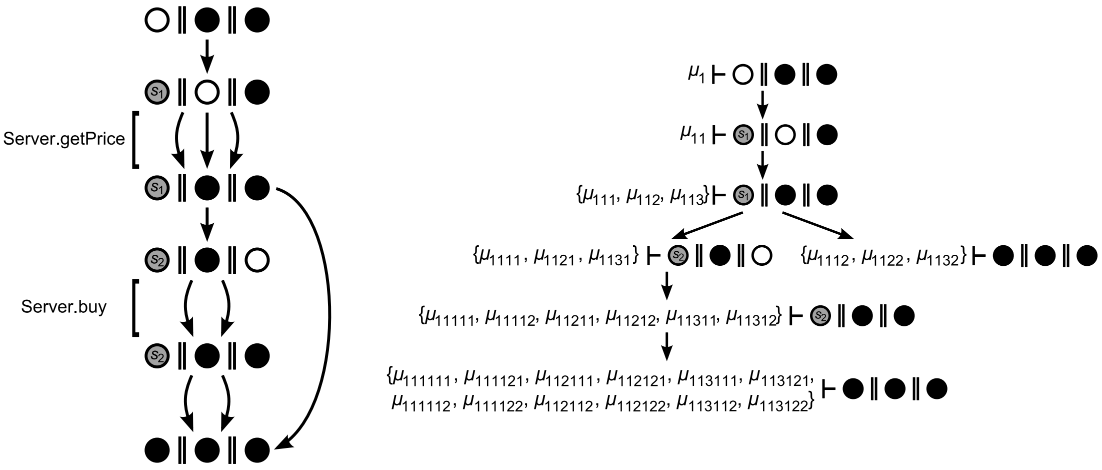
Let us write $\mathcal{X}_{args} = \{X_1,\ldots,X_n\}$ for the set of argument variables. For each argument X_k, we let the user specify its mean $Mean(X_k)$, variance $Var(X_k)$, and truncation interval $[Lower(X_k), Upper(X_k)]$. In practice, in an implementation of our analysis, we would need to provide an interface with which the user can enter this information — for example, through a dialogue box, by annotating their code, or by importing the information from an external source, such as an annotated UML use case diagram. Once we have this information, we can construct an initial measure $\mu_I^\sharp$ as follows:

$$\mu_I^\sharp = T[\boldsymbol{a},\boldsymbol{b}]N_N(\boldsymbol{\mu},\boldsymbol{\Sigma}) \tag{4.3}$$

where

$$\begin{aligned}
\boldsymbol{a}(i) &= \begin{cases} Lower(\boldsymbol{X}(i)) & \text{if } \boldsymbol{X}(i) \in \mathcal{X}_{args} \\ \bot & \text{otherwise} \end{cases} \\
\boldsymbol{b}(i) &= \begin{cases} Upper(\boldsymbol{X}(i)) & \text{if } \boldsymbol{X}(i) \in \mathcal{X}_{args} \\ \top & \text{otherwise} \end{cases} \\
\boldsymbol{\mu}(i) &= \begin{cases} Mean(\boldsymbol{X}(i)) & \text{if } \boldsymbol{X}(i) \in \mathcal{X}_{args} \\ 0 & \text{otherwise} \end{cases} \\
\boldsymbol{\Sigma}(i,j) &= \begin{cases} Var(\boldsymbol{X}(i)) & \text{if } \boldsymbol{X}(i) \in \mathcal{X}_{args} \wedge i = j \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

We initialise non-arguments to have a mean and variance of zero, and a truncation interval of $[\bot,\top]$. This is a decision that we make, on the basis that we would expect


Server.getPrice
Server.buy
$\mu_1 \vdash$
$\mu_{11} \vdash$
$\{\mu_{111}, \mu_{112}, \mu_{113}\} \vdash$
$\{\mu_{1111}, \mu_{1121}, \mu_{1131}\} \vdash$
$\{\mu_{1112}, \mu_{1122}, \mu_{1132}\} \vdash$
$\{\mu_{11111}, \mu_{11112}, \mu_{11211}, \mu_{11212}, \mu_{11311}, \mu_{11312}\} \vdash$
$\{\mu_{111111}, \mu_{111121}, \mu_{112111}, \mu_{112121}, \mu_{113111}, \mu_{113121}, \mu_{111112}, \mu_{111122}, \mu_{112112}, \mu_{112122}, \mu_{113112}, \mu_{113122}\} \vdash$

(a) System-level stages

(b) System-level abstract interpretation

Figure 4.2: Abstract interpretation of the client-server example from Figure 3.1

variables to be initialised before they are used — in this case, any choice would be equally good, as it will be overwritten. There is nothing in the Siril language that prevents an uninitialised variable from being used, however, and so our choice here can affect the behaviour of a program. Note that we could also have chosen the truncation interval to be $[0,0]$, since a variance of zero implies a point measure — this would result in an equivalent initial measure to the above.

For the argument variables, we assume that they are independent, in that all the covariances are zero. We could easily generalise this, however, by allowing the user to specifying the initial covariances between the arguments in addition to their variances.

To illustrate this progress in more detail, let us return to the client-server example from the previous chapter. The program was given in Figure 3.1, and consists of three methods: `Client.buy`, `Server.getPrice` and `Server.buy`. Of these, only `Client.buy` performs any remote procedure calls, and the concrete semantics of this method was shown in Figure 3.5. This means that the probabilistic automaton for `Client.buy` contains four stages, whereas those for `Server.getPrice` and `Server.buy` each contain only two (○ and ●).

Figure 4.2 (a) shows the transitions between the system-level stages in the program, where the position in the parallel composition indicates the method that each stage belongs to: `Client.buy` || `Server.getPrice` || `Server.buy`. The initial stage is ○ || ● || ●, which corresponds to instantiating the program with a call to `Client.buy`. The transitions corresponding to `Server.getPrice` and `Server.buy` are as labelled,

	`Client.buy`	`Server.getPrice`	`Server.buy`
$Vars_{arg}$	`quantity` `cash`	`quantity`	`quantity`

$Vars_{loc}$	`price`
	⟨`price − cash`⟩
	`success`
			`max_order`
			⟨`quantity − max_order`⟩

Figure 4.3: Variables in the client-server example from Figure 3.1

and all other transitions come from the probabilistic automaton of `Client.buy`.

Figure 4.2 (b) shows the system-level abstract interpretation of the program. Here we use the shortand notation $\{\mu_1,\ldots,\mu_n\} \vdash S$ to account for the set of states $\mu_1 \vdash S,\ldots,\mu_n \vdash S$. The subscript for each measure indicates the sequence of transitions taken, and there is a transition from $\mu_{i_1\cdots i_{n-1}} \vdash S$ to $\mu_{i_1\cdots i_{n-1}i_n} \vdash S'$ if there is a shorthand transition $M \vdash S \rightarrow M' \vdash S'$ in the figure such that $\mu_{i_1\cdots i_{n-1}} \in M$ and $\mu_{i_1\cdots i_{n-1}i_n} \in M'$. Note that we omit the superscript '$\sharp$' on these measures — since they are all truncated multivariate normal, we will not confuse the concrete and abstract domains.

Before describing the measures in this abstract interpretation in more detail, we need to be quite specific about the variables used in the program, since this determines the dimension of the truncated multivariate normal measures that we use. Figure 4.3 lists all the variables used by each method, where $Vars_{arg}$ refers to its argument variables, and $Vars_{loc}$ to its local variables. There are two things to point out here:

1. Although all three methods have an argument variable called `quantity`, these are *different variables*, and should technically be distinguished by alpha-renaming before carrying out the program analysis.

2. The variables ⟨`price − cash`⟩ and ⟨`quantity − max_order`⟩ are not present in the original program, but are needed in order to compare two variables — in this case, the comparisons '`price` ≤ `cash`' and '`quantity` ≤ `max_order`' respectively. Recall that the actual syntax of Siril only allows a variable to be compared to a constant.

There are a total of twelve variables in the program — nine are shown in the above figure, and there is additionally a return variable for each method. To avoid writing out twelve-by-twelve matrices, however, we will cheat a little by observing that the variable `quantity` is only copied between the methods, and never changes, and that the return values of `Server.getPrice` and `Server.buy` are assigned without modification to `price` and `success` respectively. We will record just one variable in each of these classes, and also omit the return variable of `Client.buy`, which is not used.

Given this, we can reduce ourselves to just seven variables, and we will specify the vector of variables $\boldsymbol{X}$ for the program as follows:

$$\begin{array}{lllllllll} \boldsymbol{X}(1) & = & \texttt{quantity} & \boldsymbol{X}(2) & = & \texttt{cash} & \boldsymbol{X}(3) & = & \texttt{price} \\ \boldsymbol{X}(4) & = & \texttt{success} & \boldsymbol{X}(5) & = & \texttt{max\_order} & \boldsymbol{X}(6) & = & \langle\texttt{price}-\texttt{cash}\rangle \\ \boldsymbol{X}(7) & = & \langle\texttt{quantity}-\texttt{max\_order}\rangle & & & & & & \end{array}$$

To execute the abstract interpretation, we first need to decide upon an initial measure for the program. We will choose the following distributions for the arguments `quantity` and `cash`:

$$\begin{array}{lcl} \texttt{quantity} & \sim & T[0,\top]N(8,9) \\ \texttt{cash} & \sim & T[0,\top]N(70,4) \end{array}$$

This is a fairly arbitrary choice, and its only real purpose is to exercise all the possibilities in the `Server.getPrice` method, so that we can see how the probabilities are calculated from the measures. If we take the above distributions, we can construct an initial measure over $\boldsymbol{X}$ as follows, using the construction in Equation 4.3:

$$\mu_I = \mu_1 = T\left(\begin{bmatrix} 0 \\ 0 \\ \perp \\ \perp \\ \perp \\ \perp \\ \perp \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

We can now proceed to compute the abstract interpretation, starting with the above initial measure, μ_1. Since Figure 4.2 (b) contains 29 measures[6] however, we will not compute all of them here, but refer the reader to Appendix B for a complete account. We will instead look at evaluating just one path in the abstract interpretation.

From the initial stage ○ || ● || ● there is just one transition leading to s_1 || ○ || ●, which corresponds to calling `Server.getPrice`. Since this is the first command in

[6] Technically, there are 35 measures, since we need two measures each for μ_{111112}, μ_{111122}, μ_{112112}, μ_{112122}, μ_{113112}, and μ_{113122} — these correspond to the condition '`success` $\neq 1$', which should be viewed as the disjunction '`success` $< 1 \vee$ `success` > 1'. We do not clutter Figure 4.2 (b) in this regard, but these are given in Appendix B where we show all the measures in detail.

Client.buy, and there are no modifications to any of the variables before the remote procedure call takes place, the measure is unchanged: $\mu_{11} = \mu_1$. The next transition is internal to Server.getPrice, and there are three possibilities depending on the value of quantity. We will consider the case when quantity > 10, which leads to the measure μ_{111} as follows:

$$\mu_{111} = T\left(\begin{bmatrix} \mathbf{10.5} \\ 0 \\ \mathbf{84} \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} \boldsymbol{\top} \\ \top \\ \boldsymbol{\top} \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ \mathbf{64} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & \mathbf{72} & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{72} & 0 & \mathbf{576} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

The changes from μ_{11} (highlighted in bold) are due to assigning '8 × quantity' to the variable price, and restricting the interval of quantity to $[10.5, \top]$. The reason for $[10.5, \top]$ and not $[10, \top]$ is that the original program was written in terms of integer-valued variables, whereas our semantics is in terms of real-valued variables. If we consider the integer value 10 to be the real interval $[9.5, 10.5]$, then the interval $[10.5, \top]$ naturally follows from the condition 'quantity $> [9.5, 10.5]$'.

From state $\mu_{111} \vdash s_1 \parallel \bullet \parallel \bullet$, the next transition is internal to Client.buy, and corresponds to the condition 'price $\leq$ cash'. When this condition is true, we move to stage $s_2 \parallel \bullet \parallel \circ$ with the following measure:

$$\mu_{1111} = T\left(\begin{bmatrix} 10.5 \\ 0 \\ 84 \\ \bot \\ \bot \\ \boldsymbol{\bot} \\ \bot \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \\ \top \\ \top \\ \mathbf{0.5} \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 64 \\ 0 \\ 0 \\ \mathbf{-6} \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 72 & 0 & 0 & \mathbf{72} & 0 \\ 0 & 4 & 0 & 0 & 0 & \mathbf{-4} & 0 \\ 72 & 0 & 576 & 0 & 0 & \mathbf{576} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{72} & \mathbf{-4} & \mathbf{576} & 0 & 0 & \mathbf{580} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

The changes here correspond to an assigment to the variable ⟨price − cash⟩, followed by a truncation due to the condition '⟨price − cash⟩ ≤ 0'.

After entering the stage $s_2 \parallel \bullet \parallel \circ$, the next transition is internal to Server.buy. Again, there are two possible transitions, but we will just consider the one corresponding to ‘`quantity` ≤ `max_order`’:

$$\mu_{11111} = T\left(\begin{bmatrix}10.5\\0\\84\\\bot\\\bot\\\bot\\\bot\end{bmatrix}, \begin{bmatrix}\top\\\top\\\top\\\top\\\top\\0.5\\\mathbf{0.5}\end{bmatrix}\right) N_7\left(\begin{bmatrix}8\\70\\64\\\mathbf{1}\\100\\-6\\\mathbf{-92}\end{bmatrix}, \begin{bmatrix}9&0&72&0&0&72&\mathbf{9}\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&\mathbf{72}\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-4&576&0&0&580&\mathbf{72}\\\mathbf{9}&0&\mathbf{72}&0&0&\mathbf{72}&\mathbf{9}\end{bmatrix}\right)$$

Here, we perform an assignment to `max_order` and to the derived variable ⟨`quantity`−`max_order`⟩, then truncate according to ‘⟨`quantity`−`max_order`⟩ ≤ 0’, before assigning 1 to `success`.

The final transition we consider is between the stages $s_2 \parallel \bullet \parallel \bullet$ and $\bullet \parallel \bullet \parallel \bullet$. For our measure μ_{11111}, only ‘`success` = 1’ will lead to a non-zero total weight, and so we choose the transition corresponding to this case:

$$\mu_{111111} = T\left(\begin{bmatrix}10.5\\\bot\\84\\\mathbf{0.5}\\\bot\\\bot\\\bot\end{bmatrix}, \begin{bmatrix}\top\\\mathbf{\top}\\\top\\\mathbf{1.5}\\\top\\0.5\\0.5\end{bmatrix}\right) N_7\left(\begin{bmatrix}8\\\mathbf{6}\\64\\1\\100\\-6\\-92\end{bmatrix}, \begin{bmatrix}9&0&72&0&0&72&9\\0&\mathbf{580}&\mathbf{-576}&0&0&\mathbf{-580}&0\\72&\mathbf{-576}&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&\mathbf{-580}&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right)$$

Here, we first perform the truncation of `success` to [0.5, 1.5], and then the assignment ‘`cash := cash - price`’.

The total weights of $\mu_1, \ldots, \mu_{111111}$ are as follows (those of the other measures in Figure 4.2 (b) can be found in Appendix B):

Measure	μ_1	μ_{11}	μ_{111}	μ_{1111}	μ_{11111}	μ_{111111}
Total Weight	0.996170	0.996170	0.202328	0.202411	0.202414	0.202414

Computing these total weights is non-trivial, as it involves integrating a multivariate normal distribution. Efficient algorithms do exist, however, such as the one described in [76], which we made use of in the form of a Java library [113].

On the basis of these total weights (along with those from Appendix B for the alternative branches), we can compute the following transitions in the abstract inter-

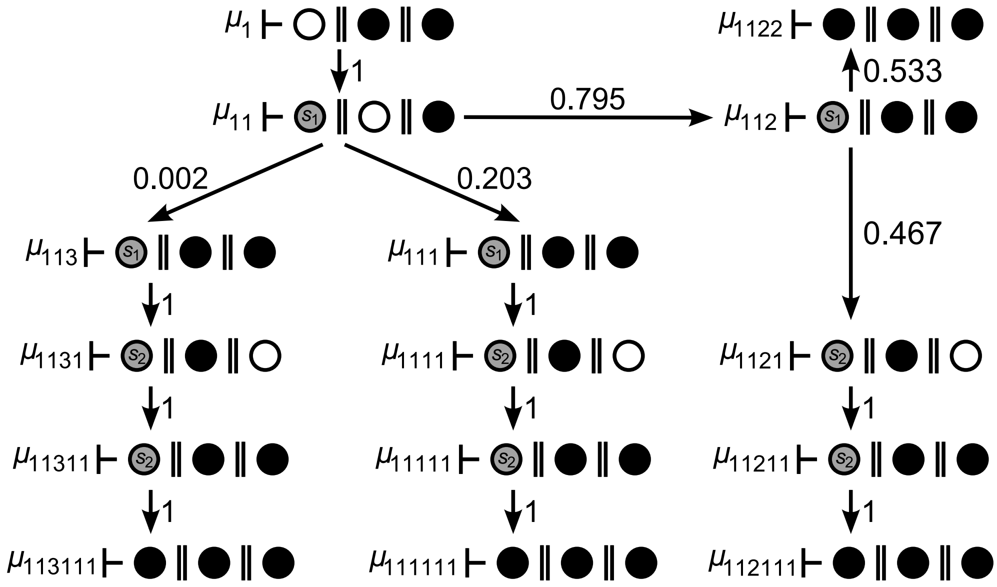


Figure 4.4: Labelled abstract interpretation of the client-server example

pretation. The probabilities are given to three decimal places:

$$\begin{array}{lllll}
\mu_1 \vdash \circ \,||\, \bullet \,||\, \bullet & \xrightarrow{1.000} & \mu_{11} \vdash s_1 \,||\, \circ \,||\, \bullet & & \\
& \xrightarrow{0.203} & \mu_{111} \vdash s_1 \,||\, \bullet \,||\, \bullet & & \\
& \xrightarrow{1.000} & \mu_{1111} \vdash s_2 \,||\, \bullet \,||\, \circ & & \\
& \xrightarrow{1.000} & \mu_{11111} \vdash s_2 \,||\, \bullet \,||\, \bullet & \xrightarrow{1.000} & \mu_{111111} \vdash \bullet \,||\, \bullet \,||\, \bullet
\end{array}$$

The value 0.203 is arrived at by taking the total weight of μ_{111} divided by the sum of the total weights of the measures at successor states of $\mu_{11} \vdash s_1 \,||\, \circ \,||\, \bullet$.

The full abstract interpretation is shown in Figure 4.4, where we exclude those states whose measure has a total weight of zero (when rounding to three decimal places), since they are effectively unreachable. To make sense of this diagram, Figure 4.5 shows the condition in the program that corresponds to each transition in the abstract interpretation. Since we have not included unreachable states, some of the conditions do not occur — for example, '`quantity` > `max_order`'.

4.5 Abstract Collecting Semantics of SIRIL

In the abstract interpretation that we just defined, a SIRIL program is treated at the system level, in that we record the state of every method at each point in time. Whilst this gives us the analysis result we intend, it is not a good basis from which to construct

Transition	Condition in Program
$\mu_1 \rightarrow \mu_{11}$	
$\mu_{11} \rightarrow \mu_{111}$	$\texttt{quantity} > 10$
$\mu_{11} \rightarrow \mu_{112}$	$0 < \texttt{quantity} \leq 10$
$\mu_{11} \rightarrow \mu_{113}$	$\texttt{quantity} \leq 0$
$\mu_{111} \rightarrow \mu_{1111}$	$\texttt{price} \leq \texttt{cash}$
$\mu_{112} \rightarrow \mu_{1121}$	$\texttt{price} \leq \texttt{cash}$
$\mu_{112} \rightarrow \mu_{1122}$	$\texttt{price} > \texttt{cash}$
$\mu_{113} \rightarrow \mu_{1131}$	$\texttt{price} \leq \texttt{cash}$
$\mu_{1111} \rightarrow \mu_{11111}$	$\texttt{quantity} \leq \texttt{max\_order}$
$\mu_{1121} \rightarrow \mu_{11211}$	$\texttt{quantity} \leq \texttt{max\_order}$
$\mu_{1131} \rightarrow \mu_{11311}$	$\texttt{quantity} \leq \texttt{max\_order}$
$\mu_{11111} \rightarrow \mu_{111111}$	$\texttt{success} = 1$
$\mu_{11211} \rightarrow \mu_{112111}$	$\texttt{success} = 1$
$\mu_{11311} \rightarrow \mu_{113111}$	$\texttt{success} = 1$

Figure 4.5: Program conditions corresponding to abstract interpretation transitions

a useful performance model. Directly going from the abstract interpretation to a continuous time Markov chain would be possible, but subsequent manipulation would be difficult, and we would lose the link between the model and the program.

The aim of this section is to present a collecting semantics that *restores the compositionality* of the original program. More specifically, we will construct a *PEPA model* from the abstract interpretation. We will do this in two parts:

1. In Section 4.5.1 we project system-level states in the abstract interpretation onto states for an individual method $O.f$. That is to say, the projected states only refer to the stages in $O.f$, and their measures only refer to the variables occurring in $O.f$. Further to this, we lift our projection so that it operates on *transitions* in the abstract interpretation. Projected transitions will have the following form:

$$\mu^\sharp \vdash s \xrightarrow{(\mu^\sharp_{in}),p,\langle\mu^\sharp_{out}\rangle} \mu'^\sharp \vdash s'$$

where $(\mu^\sharp_{in})$ is a *guard*, corresponding to 'receiving' a measure $\mu^\sharp_{in}$, and $\langle\mu^\sharp_{out}\rangle$ is an *effect*, corresponding to 'outputting' a measure $\mu^\sharp_{out}$. In this way, we collect information about the interaction points within our projected transition system.

Since we will be dealing exclusively with abstract measures in this section, we will subsequently write just μ in place of $\mu^\sharp$.

2. In Section 4.5.2 we use these projected transitions to construct a sequential PEPA component for every method. Intuitively, the guards correspond to activities with passive rates, and the effects correspond to those with active rates. We then combine the components for each method $O.f$ of an object O into a single sequential component for the entire object, so that it is viewed as a single resource in the model. Finally, we construct the system equation for the PEPA model.

An important property of the collecting semantics is that the embedded DTMC of the underlying CTMC of the PEPA model we generate is the same as the DTMC given by the abstract interpretation.

4.5.1 Projection from the System onto a Method

The first stage of our collecting semantics is to define a mapping from states of the system onto states of individual methods. For each method $O.f$, we want to have states of the form $\mu \vdash s$, where $s \in [\![O.f]\!]^{S\sharp}_{pa}$, and μ is a truncated multivariate normal measure over *only* those variables in $O.f$. We will do this by defining a projection operator $\pi_{O.f}$, which maps each system state onto a state of $O.f$. First, however, we need to formally define the set of variables that occur in a method.

Given a Siril command C in the context of a method $O.f$, the set of variable definitions $\mathit{Vars}_{O.f}(C)$ can be computed recursively as follows:

$$\begin{aligned}
\mathit{Vars}_{O.f}(\mathtt{skip}) &= \{\,\} \\
\mathit{Vars}_{O.f}(\mathtt{return}\ E) &= \{\,\} \\
\mathit{Vars}_{O.f}(X := E) &= \{X\} \\
\mathit{Vars}_{O.f}(X := f'(X_1,\ldots,X_n)) &= \{X\} \cup \mathit{Vars}_{loc}(O.f') \\
\mathit{Vars}_{O.f}(X := O'.f'(X_1,\ldots,X_n)) &= \{X\} \\
\mathit{Vars}_{O.f}(C_1\ ;\ C_2) &= \mathit{Vars}_{O.f}(C_1) \cup \mathit{Vars}_{O.f}(C_2) \\
\mathit{Vars}_{O.f}(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) &= \mathit{Vars}_{O.f}(C_1) \cup \mathit{Vars}_{O.f}(C_2) \\
\mathit{Vars}_{O.f}(\mathtt{pif}\ q\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) &= \mathit{Vars}_{O.f}(C_1) \cup \mathit{Vars}_{O.f}(C_2)
\end{aligned}$$

The only notable cases here are for method calls — we do not examine remote calls, but we include the local variables of local calls. Vars_{loc} will be defined momentarily.

We will additionally define two ways of extracting sets of variables from method definitions — Vars_{loc}, which extracts the variables defined in the body of the method,

Vars_{arg}, which extracts the argument variables of the method, and Vars_{int}, which extracts its interface variables (the argument and return variables):

$$\begin{aligned}\mathit{Vars}_{arg}(O.f(X_1,\ldots,X_n)\,\{C\}) &= \{X_1,\ldots X_n\}\\ \mathit{Vars}_{int}(O.f(X_1,\ldots,X_n)\,\{C\}) &= \{X_{O.f},X_1,\ldots X_n\}\\ \mathit{Vars}_{loc}(O.f(X_1,\ldots,X_n)\,\{C\}) &= \mathit{Vars}_{O.f}(C)\end{aligned}$$

Finally, to simplify matters, we will use the following shorthand notations, making use of the function *def* defined in the previous chapter. Recall that $\mathit{def}(O.f')$ refers to the static definition of $O.f'$, which has the form $O.f'(X_1,\ldots,X_n)\,\{C\}$:

$$\begin{aligned}\mathit{Vars}_{arg}(O.f) &= \mathit{Vars}_{arg}(\mathit{def}(O.f))\\ \mathit{Vars}_{int}(O.f) &= \mathit{Vars}_{int}(\mathit{def}(O.f))\\ \mathit{Vars}_{loc}(O.f) &= \mathit{Vars}_{loc}(\mathit{def}(O.f))\\ \mathit{Vars}(O.f) &= \mathit{Vars}_{int}(O.f)\cup \mathit{Vars}_{loc}(O.f)\end{aligned}$$

It is important to note that $\mathit{Vars}(O.f)$ defines a *set* of variables, whereas for the purposes of defining our measures, we use a *vector* of variables. This is because truncated multivariate normal measures are defined using vectors and matrices, so we need to map variables onto indices by choosing an appropriate ordering. Recall that we use $\boldsymbol{X}$ to denote the vector of N variables for the entire Siril program. Given a set of variables $\mathcal{X}\subseteq\{\boldsymbol{X}(i)\mid 1\le i\le N\}$ we can uniquely define a new vector $\boldsymbol{X}'$ of length $|\mathcal{X}|$ by the following conditions:

$$\begin{aligned}\forall 1\le i\le|\mathcal{X}|.&\quad \boldsymbol{X}'(i)\in\mathcal{X}\\ \forall 1\le i<j\le|\mathcal{X}|.&\quad \boldsymbol{X}'(i)\neq\boldsymbol{X}'(j)\wedge\iota_{\boldsymbol{X}}(\boldsymbol{X}'(i))<\iota_{\boldsymbol{X}}(\boldsymbol{X}'(j))\end{aligned}$$

We will refer to $\boldsymbol{X}'$ as the *order-preserving* vector of $\mathcal{X}$ with respect to $\boldsymbol{X}$. Although this is the most natural ordering, as it preserves that of the original vector $\boldsymbol{X}$, observe that we could equally choose any other ordering — since the purpose of this vector is to enable us to project a truncated multivariate normal measure onto a subset of its variables, a different ordering just corresponds to a permutation of the mean vector, covariance matrix, and truncation vectors.

Let us now define a projection function $\pi_{\boldsymbol{X}'}$, which takes a truncated multivariate normal measure, and projects it onto just those variables in $\boldsymbol{X}'$. We will write $\mathcal{N}^{\boldsymbol{X}'}_{|\boldsymbol{X}'|}(\boldsymbol{\mu}',\boldsymbol{\Sigma}')$ to mean the multivariate normal measure $\mathcal{N}_{|\boldsymbol{X}'|}(\boldsymbol{\mu}',\boldsymbol{\Sigma}')$, labelled with the vector of variables $\boldsymbol{X}'$ that it refers to. This label has no effect on the semantics of the measure, but is used when comparing measures — two measures are the same iff they have the same mean vector, covariance matrix, truncation interval, scaling factor *and* label.

$$\pi_{X'}(\mu_T T[\boldsymbol{a},\boldsymbol{b}] N_N(\boldsymbol{\mu},\boldsymbol{\Sigma})) = \mu_T T[\boldsymbol{a}',\boldsymbol{b}'] N^{X'}_{|X'|}(\boldsymbol{\mu}',\boldsymbol{\Sigma}') \tag{4.4}$$

where

$$\begin{array}{rclcrcl} \boldsymbol{a}'(i) & = & \boldsymbol{a}(\iota_X(\boldsymbol{X}'(i))) & \quad & \boldsymbol{\mu}'(i) & = & \boldsymbol{\mu}(\iota_X(\boldsymbol{X}'(i))) \\ \boldsymbol{b}'(i) & = & \boldsymbol{b}(\iota_X(\boldsymbol{X}'(i))) & \quad & \boldsymbol{\Sigma}'(i,j) & = & \boldsymbol{\Sigma}(\iota_X(\boldsymbol{X}'(i)),\iota_X(\boldsymbol{X}'(j))) \end{array}$$

If μ is a truncated multivariate normal *distribution*, then $\pi_{X'}(\mu)$ gives the *marginal distribution* over the variables in $\boldsymbol{X}'$. We can similarly define a projection function over stages of the system as follows:

$$\pi_{O_i.f_j}(s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{m,n_m}) = s_{i,j}$$

Finally, let us bring these two together to define a projection function $\pi_{O_i.f_j}$ over states. A state of the system, $\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{m,n_m}$, is projected onto a state of the method $O_i.f_j$ as follows:

$$\begin{array}{rcl} \pi_{O_i.f_j}(\mu \vdash s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{m,n_m}) & = & \pi_{X'}(\mu) \vdash \pi_{O_i.f_j}(s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{m,n_m}) \\ & = & \pi_{X'}(\mu) \vdash s_{i,j} \end{array}$$

where $\boldsymbol{X}'$ is the order-preserving vector of $\mathit{Vars}(O_i.f_j)$ with respect to $\boldsymbol{X}$.

This allows us to project *states* of the system onto those of a method, but it does little to help us with *transitions*. As an example, consider a SIRIL program with just three objects, and one method per object, such that the system is of the form $O_1.f \parallel O_2.f \parallel O_3.f$. If $O_1.f$ just calls $O_2.f$ and returns, and $O_2.f$ just calls $O_3.f$ and returns (and $O_3.f$ contains no branching instructions or method calls), then the abstract interpretation will look like the following:

$$\begin{array}{rcl} \mu_I \vdash \circ \parallel \bullet \parallel \bullet & \xrightarrow{p_1} & \mu_1 \vdash s_1[O_2.f] \parallel \circ \parallel \bullet \\ & \xrightarrow{p_2} & \mu_2 \vdash s_1[O_2.f] \parallel s_2[O_3.f] \parallel \circ \\ & \xrightarrow{p_3} & \mu_3 \vdash s_1[O_2.f] \parallel s_2[O_3.f] \parallel \bullet \\ & \xrightarrow{p_4} & \mu_4 \vdash s_1[O_2.f] \parallel \bullet \parallel \bullet \quad \xrightarrow{p_5} \quad \mu_5 \vdash \bullet \parallel \bullet \parallel \bullet \end{array}$$

Here, we omit the subscripts for ○ and ● since the methods they belong to are clear from their position in the parallel composition. Note that in this simple case, all the probabilities $p_1,\ldots,p_5$ will be one, since no other transitions are possible. However, we could also view this as a fragment of the abstract interpretation of a more complicated program, where other transitions are possible from each state.

Let us consider what happens when we apply the projection function $\pi_{O_1.f}$ to each of the above states. Letting $\boldsymbol{X}_1$ be the order-preserving vector of $\mathit{Vars}(O_1.f)$ with respect to $\boldsymbol{X}$, we have the following transitions:

$$\begin{tikzcd} \pi_{X_1}(\mu_I) \vdash \bigcirc \arrow[r] & \pi_{X_1}(\mu_1) \vdash s_1[O_2.f] \arrow[r] \arrow[loop below] & \pi_{X_1}(\mu_5) \vdash \bullet \end{tikzcd} \tag{4.5}$$

There are two important observations to make here. Firstly, $\pi_{X_1}(\mu_1) = \pi_{X_1}(\mu_2) = \ldots = \pi_{X_1}(\mu_4)$, since the transitions when $O_1.f$ is blocked in stage s_1 only involve variables in other methods. This means that there are two possible transitions out of the state $\pi_{X_1}(\mu_1) \vdash s_1[O_2.f]$ — one is a loop corresponding to blocking, and the other is a transition to a different state, corresponding to the remote call returning. Secondly, we have not labelled the transitions — it is no longer meaningful to use a probability, since we have introduced a choice between two transitions when there previously was none.

The fact that there is such a choice seems to imply that we have introduced non-determinism into our transition system. This is true to a certain extent, since the choice of transition is made externally of $O_1.f$. Note that this is the problem that we discussed in Section 3.5, and are now in a position to properly address.

The simplest approach we could take to avoid non-determinism would be to construct a crude approximation by taking an "average" probability for making each transition. For example, here this would give us a probability of $\frac{3}{4}$ for blocking, and $\frac{1}{4}$ for moving to the state $\pi_{X_1}(\mu_5) \vdash \bullet$. Such an approach, however, implies that we are modelling each method independently — this is not the case here, since we wish to construct a performance model of the *entire system*. The purpose of our projection is not to throw away the information about the other methods, but to regain *compositionality*. Hence, we will construct a PEPA model in which the non-determinism is resolved by cooperating with another component. Before we do so in the next subsection, however, we need to capture some information about the argument and return values of a method call. We will do this by lifting our projection operator to *transitions*.

To make the mapping into a PEPA model as easy as possible, we will introduce projected transitions of the following form:

$$\mu \vdash s \xrightarrow{(\mu_{in}),p,\langle\mu_{out}\rangle} \mu' \vdash s'$$

This means that *if we input* the measure μ_{in}, we take the transition with probability p, and in doing so *output* the measure μ_{out}. Due to the nature of our probabilistic automaton semantics, every stage corresponds to a remote method call, and so we will always have an input and output measure on every transition.

Intuitively, a method will *input* a measure when it is invoked, or when it receives a return value and is unblocked. It will *output* a measure when it calls another method, or when it returns a value. It is important to realise, however, that this notion of input and output is abstract, since it does not affect the measure μ' after taking the transition. μ' has already been modified in the way we expect by the system-level abstract interpretation itself. Hence these inputs and outputs are simply used as labels, to enable us to match synchronisation points in different methods.

We can now lift the projection operator $\pi_{O.f}$ to induce a transition system of the above form as follows, where we abbreviate a system stage $s_{1,1} \parallel \cdots \parallel s_{i,j} \parallel \cdots \parallel s_{m,n_m}$ to just S for brevity:

$$\pi_{O.f}(\mu \vdash S \xrightarrow{p} \mu' \vdash S') = \begin{cases} \{\,\} & \text{if } \pi_{O.f}(\mu \vdash S) = \pi_{O.f}(\mu' \vdash S') \vee S = \bullet_{O.f} \\ \{\pi_{O.f}(\mu \vdash S) \xrightarrow{\kappa_{in}, p, \kappa_{out}} \pi_{O.f}(\mu' \vdash S')\} & \text{otherwise} \end{cases} \tag{4.6}$$

where

$$\kappa_{in} = \begin{cases} (\pi_{Vars_{arg}(O.f)}(\mu)) & \text{if } \pi_{O.f}(S) = \circ_{O.f} \\ (\pi_{Vars_{int}(O'.f')}(\mu)) & \text{if } \pi_{O.f}(S) = s[O'.f'] \end{cases}$$
$$\kappa_{out} = \begin{cases} \langle \pi_{Vars_{int}(O.f)}(\mu') \rangle & \text{if } \pi_{O.f}(S') = \bullet_{O.f} \\ \langle \pi_{Vars_{arg}(O'.f')}(\mu') \rangle & \text{if } \pi_{O.f}(S') = s[O'.f'] \end{cases}$$

When we use $Vars_{int}(O.f)$ in the above, we really mean the order-preserving vector of $Vars_{int}(O.f)$ with respect to $\boldsymbol{X}$. The reason for returning a *set* of transitions is to allow us to remove self-loops, which will not contribute to the PEPA model we generate. We also remove transitions out of the terminal stages $\bullet_{O.f}$, as we shall explicitly deal with instantiating methods when we map onto PEPA in the next section.

To understand this, let us look at the case of κ_{in}. If the transition is from the initial stage $\circ_{O.f}$, then we input the measure on the argument variables of $O.f$ before executing the transition — i.e. the initial values of the method's arguments. If the transition is from an intermediate stage $s[O'.f']$, however, we need to input the return value of the method $O'.f'$. But in order for $O.f$ to be able to make a transition, $O'.f'$ must have already terminated, and so we look at the interface variables of $O'.f'$ before executing the transition. Note that it is not sufficient in this latter case to look at only the return variable $X_{O'.f'}$, since we would be unable to distinguish between two instances where we return the same value (in terms of mean and variance), but have different covariances with respect to the arguments we called the method with.

If we view our abstract interpretation to be a set of transitions T, each of the form

$\mu \vdash S \xrightarrow{p} \mu' \vdash S'$, then our collecting semantics for the method $O.f$ is as follows:

$$Coll_{O.f}(T) = \bigcup_{t \in T} \pi_{O.f}(t)$$

Returning to our example from Equation 4.5, we can now label the transitions:

$$\pi_{X_1}(\mu_I) \vdash \bigcirc \xrightarrow{(\mu_{call_1}), p_1, \langle \mu_{call_2} \rangle} \pi_{X_1}(\mu_1) \vdash s_1[O_2.f] \xrightarrow{(\mu_{ret_2}), p_5, \langle \mu_{ret_1} \rangle} \pi_{X_1}(\mu_5) \vdash \bullet$$

Notice that the self-loop on the second state has disappeared, but the possibility of blocking is still present due to the guards on the transitions. The values of μ_{call_2} and μ_{ret_2} depend on the actual measures in the abstract interpretation of the program, but are synchronised with those in the collected transitions for $O_2.f$:

$$\pi_{X_2}(\mu_1) \vdash \bigcirc \xrightarrow{(\mu_{call_2}), p_2, \langle \mu_{call_3} \rangle} \pi_{X_2}(\mu_2) \vdash s_2[O_3.f] \xrightarrow{(\mu_{ret_3}), p_4, \langle \mu_{ret_2} \rangle} \pi_{X_2}(\mu_4) \vdash \bullet$$

where X_2 is the order-preserving vector of $Vars(O_2.f)$ with respect to X. Finally, for $O_3.f$, there is only one collected transition:

$$\pi_{X_3}(\mu_2) \vdash \bigcirc \xrightarrow{(\mu_{call_3}), p_3, \langle \mu_{ret_3} \rangle} \pi_{X_3}(\mu_3) \vdash \bullet$$

where X_3 is the order-preserving vector of $Vars(O_3.f)$ with respect to X.

Returning to our client-server example, we can project the abstract interpretation shown in Figure 4.4 by directly applying Equation 4.6. We will use $C.B$ as shorthand for `Client.buy`, and $S.P$ and $S.B$ for `Server.getPrice` and `Server.buy` respectively. To illustrate how this is done, let us examine the projection of the first transition in Figure 4.4 onto `Client.buy`. Recall that the system-level transition is:

$$\mu_1 \vdash \bigcirc \,||\, \bullet \,||\, \bullet \xrightarrow{1} \mu_{11} \vdash s_1[S.P] \,||\, \bigcirc \,||\, \bullet$$

Applying Equation 4.6, we get the following (we define κ_{in} and κ_{out} momentarily):

$$\pi_{C.B}(\mu_1) \vdash \bigcirc \xrightarrow{\kappa_{in}, 1, \kappa_{out}} \pi_{C.B}(\mu_{11}) \vdash s_1[S.P]$$

Here, we use $\pi_{C.B}(\mu)$ to mean $\pi_{X'}(\mu)$ where X' is the order-preserving vector of $Vars(C.B)$ with respect to X. In our case, X' is a vector of length 5, such that:

$$\begin{array}{lllllllll} X'(1) & = & \texttt{quantity} & X'(2) & = & \texttt{cash} & X'(3) & = & \texttt{price} \\ X'(4) & = & \texttt{success} & X'(5) & = & \langle \texttt{price} - \texttt{cash} \rangle & & & \end{array}$$

$$
\begin{array}{rcl}
\pi_{C.B}(\mu_1) \vdash \bigcirc & \xrightarrow{(\pi_{Vars_{arg}(C.B)}(\mu_1)),1,\langle\pi_{Vars_{arg}(S.P)}(\mu_{11})\rangle} & \pi_{C.B}(\mu_{11}) \vdash s_1 \\
\pi_{C.B}(\mu_{11}) \vdash s_1 = \pi_{C.B}(\mu_{111}) \vdash s_1 & \xrightarrow{(\pi_{Vars_{int}(S.P)}(\mu_{111})),1,\langle\pi_{Vars_{arg}(S.B)}(\mu_{1111})\rangle} & \pi_{C.B}(\mu_{1111}) \vdash s_2 \\
\pi_{C.B}(\mu_{11}) \vdash s_1 = \pi_{C.B}(\mu_{112}) \vdash s_1 & \xrightarrow{(\pi_{Vars_{int}(S.P)}(\mu_{112})),0.476,\langle\pi_{Vars_{arg}(S.B)}(\mu_{1121})\rangle} & \pi_{C.B}(\mu_{1121}) \vdash s_2 \\
\pi_{C.B}(\mu_{11}) \vdash s_1 = \pi_{C.B}(\mu_{112}) \vdash s_1 & \xrightarrow{(\pi_{Vars_{int}(S.P)}(\mu_{112})),0.533,\langle\pi_{Vars_{int}(C.B)}(\mu_{1122})\rangle} & \pi_{C.B}(\mu_{1122}) \vdash \bullet \\
\pi_{C.B}(\mu_{11}) \vdash s_1 = \pi_{C.B}(\mu_{113}) \vdash s_1 & \xrightarrow{(\pi_{Vars_{int}(S.P)}(\mu_{113})),1,\langle\pi_{Vars_{arg}(S.B)}(\mu_{1131})\rangle} & \pi_{C.B}(\mu_{1131}) \vdash s_2 \\
\pi_{C.B}(\mu_{1111}) \vdash s_2 = \pi_{C.B}(\mu_{11111}) \vdash s_2 & \xrightarrow{(\pi_{Vars_{int}(S.B)}(\mu_{11111})),1,\langle\pi_{Vars_{int}(C.B)}(\mu_{111111})\rangle} & \pi_{C.B}(\mu_{111111}) \vdash \bullet \\
\pi_{C.B}(\mu_{1121}) \vdash s_2 = \pi_{C.B}(\mu_{11211}) \vdash s_2 & \xrightarrow{(\pi_{Vars_{int}(S.B)}(\mu_{11211})),1,\langle\pi_{Vars_{int}(C.B)}(\mu_{112111})\rangle} & \pi_{C.B}(\mu_{112111}) \vdash \bullet \\
\pi_{C.B}(\mu_{1131}) \vdash s_2 = \pi_{C.B}(\mu_{11311}) \vdash s_2 & \xrightarrow{(\pi_{Vars_{int}(S.B)}(\mu_{11311})),1,\langle\pi_{Vars_{int}(C.B)}(\mu_{113111})\rangle} & \pi_{C.B}(\mu_{113111}) \vdash \bullet
\end{array}
$$

$$
\begin{array}{rcl}
\pi_{S.P}(\mu_{11}) \vdash \bigcirc & \xrightarrow{(\pi_{Vars_{arg}(S.P)}(\mu_{11})),0.203,\langle\pi_{Vars_{int}(S.P)}(\mu_{111})\rangle} & \pi_{S.P}(\mu_{111}) \vdash \bullet \\
\pi_{S.P}(\mu_{11}) \vdash \bigcirc & \xrightarrow{(\pi_{Vars_{arg}(S.P)}(\mu_{11})),0.795,\langle\pi_{Vars_{int}(S.P)}(\mu_{112})\rangle} & \pi_{S.P}(\mu_{112}) \vdash \bullet \\
\pi_{S.P}(\mu_{11}) \vdash \bigcirc & \xrightarrow{(\pi_{Vars_{arg}(S.P)}(\mu_{11})),0.002,\langle\pi_{Vars_{int}(S.P)}(\mu_{113})\rangle} & \pi_{S.P}(\mu_{113}) \vdash \bullet
\end{array}
$$

$$
\begin{array}{rcl}
\pi_{S.B}(\mu_{1111}) \vdash \bigcirc & \xrightarrow{(\pi_{Vars_{arg}(S.B)}(\mu_{1111})),1,\langle\pi_{Vars_{int}(S.B)}(\mu_{11111})\rangle} & \pi_{S.B}(\mu_{11111}) \vdash \bullet \\
\pi_{S.B}(\mu_{1121}) \vdash \bigcirc & \xrightarrow{(\pi_{Vars_{arg}(S.B)}(\mu_{1121})),1,\langle\pi_{Vars_{int}(S.B)}(\mu_{11211})\rangle} & \pi_{S.B}(\mu_{11211}) \vdash \bullet \\
\pi_{S.B}(\mu_{1131}) \vdash \bigcirc & \xrightarrow{(\pi_{Vars_{arg}(S.B)}(\mu_{1131})),1,\langle\pi_{Vars_{int}(S.B)}(\mu_{11311})\rangle} & \pi_{S.B}(\mu_{11311}) \vdash \bullet
\end{array}
$$

Figure 4.6: Projected abstract interpretation of the client-server example

As an example, $\pi_{C.B}(\mu_1)$ results in the following projected measure:

$$\pi_{C.B}(\mu_1) = T\left(\begin{bmatrix} 0 \\ 0 \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_5 \left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

We now just need to find the values of κ_{in} and κ_{out} with which to label our transition. Since the initial stage is $\bigcirc_{C.B}$, κ_{in} will be $(\pi_{Vars_{arg}(C.B)}(\mu_1))$, which corresponds to invoking `Client.buy` with the arguments given by μ_1. Similarly, the target stage is $s[S.P]$, and so κ_{out} will be $\langle\pi_{Vars_{arg}(S.P)}(\mu_{11})\rangle$, which corresponds to invoking `Server.getPrice` with the arguments given by μ_{11}. Putting these directly onto the projected transition, we get:

$$\pi_{C.B}(\mu_1) \vdash \bigcirc \xrightarrow{(\pi_{Vars_{arg}(C.B)}(\mu_1)),1,\langle\pi_{Vars_{arg}(S.P)}(\mu_{11})\rangle} \pi_{C.B}(\mu_{11}) \vdash s_1$$

This transition can be read informally as "if we are in state $\pi_{C.B}(\mu_1) \vdash \bigcirc$ and `Client.buy` is invoked with the arguments μ_1, then with probability 1 we will invoke `Server.getPrice` with the arguments μ_{11} and move to state $\pi_{C.B}(\mu_{11}) \vdash s_1$".

The complete set of projected transitions for each method is shown in Figure 4.6. The labels on stages s_1 and s_2 are omitted to save space. The construction of each of these transitions follows exactly the same pattern as above, but it is important to note that the equalities between states — for example, $\pi_{C.B}(\mu_{11}) \vdash s_1 = \pi_{C.B}(\mu_{111}) \vdash s_1$ — refer to the measures and projections over the full vector of variables for the program (i.e. all twelve variables as opposed to just seven). This is the only real consequence of 'cheating' by eliminating certain variables that appear to be redundant — everything else, including the total weight of the measure, is the same for the reduced set of variables.

4.5.2 Construction of the PEPA Model

Given a collected transition system as defined in the previous section, it is now straightforward to map onto PEPA components. We will do this in three stages — first constructing a sequential component for each method $O.f$, then combining these to create one component per object O, and finally constructing the system equation.

It is important to realise that at this stage in the analysis, the semantics of measures becomes unimportant. The transitions are already labelled with probabilities due to the abstract interpretation, and so the measures have no other purpose that to act as *labels*, which we will use in constructing PEPA activity types. Further to this, we shall refer to states $\mu \vdash s$ by just a single meta-variable S, as there is no longer any need to 'look inside' at the actual measure.

Before we define our mapping onto PEPA, let us introduce some useful notation, which allows us to collect the set of transitions corresponding to a particular starting state S and input measure κ_{block}. We do so using a function $Trans_{O.f}(S,\kappa_{block})$, and we additionally define the predicate $Input_{O.f}(S,\kappa_{block})$ which evaluates to true iff κ_{block} is a possible input in state S:

$$\begin{array}{rcl} Trans_{O.f}(S,\kappa_{block}) & = & \{S_1 \xrightarrow{\kappa_{in},p,\kappa_{out}} S_2 \in Coll_{O.f}(T) \mid S = S_1 \wedge \kappa_{block} = \kappa_{in}\} \\ Input_{O.f}(S,\kappa_{block}) & = & Trans_{O.f}(S,\kappa_{block}) \neq \emptyset \end{array}$$

where T is the set of transitions corresponding to the abstract interpretation of the program.

We will define a sequential PEPA state for every state S that occurs in the collected abstract interpretation of the method $O.f$. As it does not matter what name we decide to give this (so long as every occurrence has the same name), we shall denote it by $State(O.f,S)$. We have a similar definition for action types, $Act(\kappa)$, defined in such a way that $Act((\mu)) = Act(\langle\mu\rangle)$. Note that Equation 4.4 ensures that the input measures to different methods will have different labels, and so will lead to different action types even if the underlying measure is the same.

For each state S in the collected abstract interpretation of the method $O.f$, we generate the following sequential state definitions in PEPA:

$$\begin{array}{rcl} State(O.f,S) & \overset{def}{=} & \displaystyle\sum_{\{\kappa_{in} \mid Input_{O.f}(S,\kappa_{in})\}} (Act(\kappa_{in}),\top).State(O.f,S,\kappa_{in}) \\ State(O.f,S,\kappa_{in}) & \overset{def}{=} & \displaystyle\sum_{\{S \xrightarrow{\kappa_{in},p,\kappa_{out}} S' \in Trans_{O.f}(S,\kappa_{in})\}} (Act(\kappa_{out}),R(O.f,p,S')).State(O.f,S') \end{array} \tag{4.7}$$

where we introduce an intermediate state $State(O.f,S,\kappa_{in})$ for each possible input κ_{in}, to make the presentation more readable. We define the rate of the output activity as:

$$R(O.f,p,S') = \begin{cases} p.((R_I(O))^{-1} + (R_E(O))^{-1})^{-1} & \text{if } S' = \mu \vdash \bullet_{O.f} \\ p.((R_I(O))^{-1} + (R_E(O'))^{-1})^{-1} & \text{if } S' = \mu \vdash s[O'.f'] \end{cases}$$

These correspond to Equations 3.8 and 3.9 of the continuous time concrete interpretation in Section 3.4.2.

Given this, we can generate a sequential PEPA component for each call to a method $O.f$ in the SIRIL program — for each initial state S of the form $\mu \vdash \circ_{O.f}$, we generate a PEPA component with initial state $State(O.f,S)$. Note, however, that all such components will lead to an absorbing state corresponding to a state in the collected abstract interpretation of the form $\mu \vdash \bullet_{O.f}$, because we do not collect transitions from the $\bullet_{O.f}$ to the $\circ_{O.f}$ stages. This is a problem, since we really want to 'reset' the method after each call, to allow it to be called again. We will fix this, however, in the process of constructing a single PEPA component for each object O in the program.

Let $Defs(O.f)$ be the set of definitions (of the form $A \overset{def}{=} C$), and $States(O.f)$ be the set of all states generated from the collected abstract interpretation of the method $O.f$. We will define the following auxilliary terms:

$$\begin{array}{lcl} Defs(O) & = & \bigcup_f Defs(O.f) \\ States(O) & = & \bigcup_f States(O.f) \\ InitDefs(O) & = & \bigcup_f \{ State(O.f,S) \overset{def}{=} C \in Defs(O.f) \mid S = \mu \vdash \circ_{O.f} \} \\ TermStates(O) & = & \bigcup_f \{ State(O.f,S) \in States(O) \mid S = \mu \vdash \bullet_{O.f} \} \end{array}$$

Here $Defs(O)$ and $States(O)$ collect together all the definitions and states corresponding to the object O, respectively. $InitDefs(O)$ is the set of all definitions corresponding to an initial stage $\circ_{O.f}$ of a method $O.f$, and $TermStates(O)$ is the set of all states corresponding to its terminal stage $\bullet_{O.f}$.

Using the above, we can construct a new set of definitions for the object O, which collects them under a single sequential component, with the initial state $State(O)$:

$$\begin{array}{lcl} NewDefs(O) & = & \left\{ State(O) \overset{def}{=} \sum_{A \overset{def}{=} C \in InitDefs(O)} C \right\} \\ & \cup & (Defs(O) \backslash InitDefs(O))[State(O)/TermStates(O)] \end{array} \quad (4.8)$$

where $[State(O)/TermStates(O)]$ should be read as substituting every occurrence of a name in $TermStates(O)$ by $State(O)$. Recall from Equation 4.4 that we label the projected measures with the variables that they refer to, which also records the method that they refer to. This means that there is no possibility of overlap between the action types of different methods belonging to the same object.

It is now straightforward to write down a system equation for our PEPA model (of a SIRIL program with M objects) as follows:

$$State(O_1) \underset{*}{\bowtie} \cdots \underset{*}{\bowtie} State(O_M)$$

where '$P \underset{*}{\bowtie} Q$' means that we synchronise over all shared action types in P and Q. More formally, this is equivalent to '$P \underset{L}{\bowtie} Q$', where L is the intersection of the sets of action types that occur in P and Q. The order in which we apply the compositions does not matter, as there are no name conflicts, or internal choices, in the model.

There is clearly a big problem with this system equation as it stands, however — it is a deadlocked process. This is because all methods are in effect 'waiting' to be called, and are blocking on passive activities. We therefore need to instantiate the model, given that the program was instantiated with a call to method $O_i.f_j$, and an initial measure μ_I.

There are two potential ways of doing this. The most direct would be to change the state corresponding to the object O_i in the system equation, so as to hard-code the call to the initial method. An alternative is to introduce an additional component corresponding to a user of the program, which invokes the method $O_i.f_j$. We will take this latter approach, because it allows more flexibility, which will be useful in the next section. The user component for invoking $O_i.f_j$ can be specified as follows:

$$\begin{array}{rcl} User_{O_i.f_j} & \overset{def}{=} & (Act(\langle\pi_{X'}(\mu_I)\rangle), r_{init}).User'_{O_i.f_j} \\ User'_{O_i.f_j} & \overset{def}{=} & \sum_{\mu \in TM}(Act((\pi_{X''}(\mu))), \top).Nil \end{array} \tag{4.9}$$

where $\boldsymbol{X}'$ is the order-preserving vector of $Vars_{arg}(O_i.f_j)$ with respect to $\boldsymbol{X}$, $\boldsymbol{X}''$ is the order-preserving vector of $Vars_{int}(O_i.f_j)$ with respect to $\boldsymbol{X}$, and r_{init} is the rate at which the user makes the initial call. TM is the set of terminal measures in the abstract interpretation — $\mu \in TM$ iff there is a state $\mu \vdash \bullet \parallel \cdots \parallel \bullet$. We define Nil to be a livelocked PEPA process, which corresponds to an absorbing state in the underlying Markov chain. This could be defined, for example, as $Nil = (\tau, r).Nil$ for any $r > 0 \in \mathbb{R}$.

This leads us finally to a correct system equation for the PEPA model of our Siril program, where the user invokes the method $O_i.f_j$:

$$User_{O_i.f_j} \underset{*}{\bowtie} State(O_1) \underset{*}{\bowtie} \cdots \underset{*}{\bowtie} State(O_M) \tag{4.10}$$

The only difference between the embedded DTMC underlying this PEPA model and that of the abstract interpretation is that we have introduced an additional state, corresponding to the user invoking the program.

Returning to our client-server example, we can now translate the projected abstract interpretation for each method into a sequential PEPA component. To do this, we first need to define the states and action types we will use. The sequential PEPA states are shown in Figure 4.7, and the action types in Figure 4.8. We will take the internal and

$$
\begin{aligned}
\mathit{State}[C.B]_1 &= \mathit{State}(\pi_{C.B}(\mu_{1}) \vdash \bigcirc) \\
\mathit{State}[C.B]_2 &= \mathit{State}(\pi_{C.B}(\mu_{11}) \vdash s_1) \\
&= \mathit{State}(\pi_{C.B}(\mu_{111}) \vdash s_1) \\
&= \mathit{State}(\pi_{C.B}(\mu_{112}) \vdash s_1) \\
&= \mathit{State}(\pi_{C.B}(\mu_{113}) \vdash s_1) \\
\mathit{State}[C.B]_3 &= \mathit{State}(\pi_{C.B}(\mu_{1122}) \vdash \bullet) \\
\mathit{State}[C.B]_4 &= \mathit{State}(\pi_{C.B}(\mu_{1111}) \vdash s_2) \\
&= \mathit{State}(\pi_{C.B}(\mu_{11111}) \vdash s_2) \\
\mathit{State}[C.B]_5 &= \mathit{State}(\pi_{C.B}(\mu_{1121}) \vdash s_2) \\
&= \mathit{State}(\pi_{C.B}(\mu_{11211}) \vdash s_2) \\
\mathit{State}[C.B]_6 &= \mathit{State}(\pi_{C.B}(\mu_{1131}) \vdash s_2) \\
&= \mathit{State}(\pi_{C.B}(\mu_{11311}) \vdash s_2) \\
\mathit{State}[C.B]_7 &= \mathit{State}(\pi_{C.B}(\mu_{111111}) \vdash \bullet) \\
\mathit{State}[C.B]_8 &= \mathit{State}(\pi_{C.B}(\mu_{112111}) \vdash \bullet) \\
\mathit{State}[C.B]_9 &= \mathit{State}(\pi_{C.B}(\mu_{113111}) \vdash \bullet)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{State}[S.P]_1 &= \mathit{State}(\pi_{S.P}(\mu_{11}) \vdash \bigcirc) \\
\mathit{State}[S.P]_2 &= \mathit{State}(\pi_{S.P}(\mu_{111}) \vdash \bullet) \\
\mathit{State}[S.P]_3 &= \mathit{State}(\pi_{S.P}(\mu_{112}) \vdash \bullet) \\
\mathit{State}[S.P]_4 &= \mathit{State}(\pi_{S.P}(\mu_{113}) \vdash \bullet)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{State}[S.B]_1 &= \mathit{State}(\pi_{S.B}(\mu_{1111}) \vdash \bigcirc) \\
\mathit{State}[S.B]_2 &= \mathit{State}(\pi_{S.B}(\mu_{1121}) \vdash \bigcirc) \\
\mathit{State}[S.B]_3 &= \mathit{State}(\pi_{S.B}(\mu_{1131}) \vdash \bigcirc) \\
\mathit{State}[S.B]_4 &= \mathit{State}(\pi_{S.B}(\mu_{11111}) \vdash \bullet) \\
\mathit{State}[S.B]_5 &= \mathit{State}(\pi_{S.B}(\mu_{11211}) \vdash \bullet) \\
\mathit{State}[S.B]_6 &= \mathit{State}(\pi_{S.B}(\mu_{11311}) \vdash \bullet)
\end{aligned}
$$

Figure 4.7: PEPA states for the client-server example

$$
\begin{aligned}
\mathit{call}[C.B] &= \mathit{Act}(\pi_{\mathit{Vars}_{arg}(C.B)}(\mu_{1})) \\
\mathit{ret}[C.B]_1 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(C.B)}(\mu_{1122})) \\
\mathit{ret}[C.B]_2 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(C.B)}(\mu_{111111})) \\
\mathit{ret}[C.B]_3 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(C.B)}(\mu_{112111})) \\
\mathit{ret}[C.B]_4 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(C.B)}(\mu_{113111}))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{call}[S.P] &= \mathit{Act}(\pi_{\mathit{Vars}_{arg}(S.P)}(\mu_{11})) \\
\mathit{ret}[S.P]_1 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(S.P)}(\mu_{111}))) \\
\mathit{ret}[S.P]_2 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(S.P)}(\mu_{112})) \\
\mathit{ret}[S.P]_3 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(S.P)}(\mu_{113}))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{call}[S.B]_1 &= \mathit{Act}(\pi_{\mathit{Vars}_{arg}(S.B)}(\mu_{1111})) \\
\mathit{call}[S.B]_2 &= \mathit{Act}(\pi_{\mathit{Vars}_{arg}(S.B)}(\mu_{1121})) \\
\mathit{call}[S.B]_3 &= \mathit{Act}(\pi_{\mathit{Vars}_{arg}(S.B)}(\mu_{1131})) \\
\mathit{ret}[S.B]_1 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(S.B)}(\mu_{11111})) \\
\mathit{ret}[S.B]_2 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(S.B)}(\mu_{11211})) \\
\mathit{ret}[S.B]_3 &= \mathit{Act}(\pi_{\mathit{Vars}_{int}(S.B)}(\mu_{11311}))
\end{aligned}
$$

Figure 4.8: PEPA action types for the client-server example

external rates of the client and server to be as follows:

$$\begin{array}{rclcrcl} R_I(\texttt{Client}) & = & 1 & \qquad & R_I(\texttt{Server}) & = & 1 \\ R_E(\texttt{Client}) & = & 0 & \qquad & R_E(\texttt{Server}) & = & \frac{1}{99} \end{array}$$

We are using a seemingly odd value for $R_E(\texttt{Server})$ so that the rate we get when we combine it with the internal rates is a round number — more specifically, activites that involve an interaction with the server over the network are 100 times slower than those that do not.

As a simple example of how we construct a PEPA sequential process definition, let us consider the initial state of the client, $State[C.B]_1$. This corresponds to the projected state $\pi_{C.B}(\mu_1) \vdash \bigcirc$ in the abstract interpretation. If we consult Figure 4.6, we can see that there is just one projected transition involving this state:

$$\pi_{C.B}(\mu_1) \vdash \bigcirc \xrightarrow{(\pi_{Vars_{arg}(C.B)}(\mu_1)),1,\langle \pi_{Vars_{arg}(S.P)}(\mu_{11})\rangle} \pi_{C.B}(\mu_{11}) \vdash s_1$$

Given that $Act((\pi_{Vars_{arg}(C.B))}(\mu_1)) = call[C.B]$, $Act(\langle \pi_{Vars_{arg}(S.P)}(\mu_{11})\rangle) = call[S.P]$ and $State(\pi_{C.B}(\mu_{11}) \vdash s_1) = State[C.B]_2$, we can construct a PEPA sequential process definition using Equation 4.7 as follows:

$$State[C.B]_1 \stackrel{def}{=} (call[C.B], \top).(call[S.P], 0.01).State[C.B]_2$$

Note that Equation 4.7 gives an explicit name for the intermediate state between carrying out the input and output activities. Because there is no choice after the input activity, however, we leave this as an implicit state in the above. The rate 0.01 of the output activity is calculated as:

$$R = p.\frac{1}{\frac{1}{R_I(\texttt{Client})} + \frac{1}{R_E(\texttt{Server})}} = 1.\frac{1}{1+99} = 0.01$$

The remaining sequential process definitions for the client-server example are shown in Figure 4.9. Notice that in `Client.buy` there are no definitions for $State[C.B]_3$, $State[C.B]_7$, $State[C.B]_8$, and $State[C.B]_9$, since these correspond to terminal states (the same is true for the terminal states of `Server.getPrice` and `Server.buy`). We resolve this by constructing object-level sequential components according to Equation 4.8, which are shown in Figure 4.10. The changes relative to Figure 4.9 are highlighted in bold.

We can now finally give our system equation, corresponding to invoking the `Client.buy` method:

$$User_{C.B} \underset{*}{\bowtie} State[C] \underset{*}{\bowtie} State[S]$$

$$
\begin{aligned}
\mathit{State}[C.B]_1 &\stackrel{\mathit{def}}{=} (\mathit{call}[C.B],\top).(\mathit{call}[S.P],0.01).\mathit{State}[C.B]_2\\
\mathit{State}[C.B]_2 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.P]_1,\top).(\mathit{call}[S.B]_1,0.01).\mathit{State}[C.B]_4\\
&\;\;+\;\; (\mathit{ret}[S.P]_2,\top).\mathit{State}[C.B]'_2\\
&\;\;+\;\; (\mathit{ret}[S.P]_3,\top).(\mathit{call}[S.B]_3,0.01).\mathit{State}[C.B]_6\\
\mathit{State}[C.B]'_2 &\stackrel{\mathit{def}}{=} (\mathit{call}[S.B]_2,0.00476).\mathit{State}[C.B]_5\\
&\;\;+\;\; (\mathit{ret}[C.B]_1,0.00533).\mathit{State}[C.B]_3\\
\mathit{State}[C.B]_4 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.B]_1,\top).(\mathit{ret}[C.B]_2,1).\mathit{State}[C.B]_7\\
\mathit{State}[C.B]_5 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.B]_2,\top).(\mathit{ret}[C.B]_3,1).\mathit{State}[C.B]_8\\
\mathit{State}[C.B]_6 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.B]_3,\top).(\mathit{ret}[C.B]_4,1).\mathit{State}[C.B]_9\\
\\
\mathit{State}[S.P]_1 &\stackrel{\mathit{def}}{=} (\mathit{call}[S.P],\top).\mathit{State}[S.P]'_1\\
\mathit{State}[S.P]'_1 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.P]_1,0.00203).\mathit{State}[S.P]_2\\
&\;\;+\;\; (\mathit{ret}[S.P]_2,0.00795).\mathit{State}[S.P]_3\\
&\;\;+\;\; (\mathit{ret}[S.P]_3,0.00002).\mathit{State}[S.P]_4\\
\\
\mathit{State}[S.B]_1 &\stackrel{\mathit{def}}{=} (\mathit{call}[S.B]_1,\top).(\mathit{ret}[S.B]_1,0.01).\mathit{State}[S.B]_4\\
\mathit{State}[S.B]_2 &\stackrel{\mathit{def}}{=} (\mathit{call}[S.B]_2,\top).(\mathit{ret}[S.B]_2,0.01).\mathit{State}[S.B]_5\\
\mathit{State}[S.B]_3 &\stackrel{\mathit{def}}{=} (\mathit{call}[S.B]_3,\top).(\mathit{ret}[S.B]_3,0.01).\mathit{State}[S.B]_6
\end{aligned}
$$

Figure 4.9: Sequential process definitions for the client-server example

$$
\begin{aligned}
\boldsymbol{\mathit{State}[C]} &\stackrel{\mathit{def}}{=} (\mathit{call}[C.B],\top).(\mathit{call}[S.P],0.01).\mathit{State}[C.B]_2\\
\mathit{State}[C.B]_2 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.P]_1,\top).(\mathit{call}[S.B]_1,0.01).\mathit{State}[C.B]_4\\
&\;\;+\;\; (\mathit{ret}[S.P]_2,\top).\mathit{State}[C.B]'_2\\
&\;\;+\;\; (\mathit{ret}[S.P]_3,\top).(\mathit{call}[S.B]_3,0.01).\mathit{State}[C.B]_6\\
\mathit{State}[C.B]'_2 &\stackrel{\mathit{def}}{=} (\mathit{call}[S.B]_2,0.00476).\mathit{State}[C.B]_5\\
&\;\;+\;\; (\mathit{ret}[C.B]_1,0.00533).\boldsymbol{\mathit{State}[C]}\\
\mathit{State}[C.B]_4 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.B]_1,\top).(\mathit{ret}[C.B]_2,1).\boldsymbol{\mathit{State}[C]}\\
\mathit{State}[C.B]_5 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.B]_2,\top).(\mathit{ret}[C.B]_3,1).\boldsymbol{\mathit{State}[C]}\\
\mathit{State}[C.B]_6 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.B]_3,\top).(\mathit{ret}[C.B]_4,1).\boldsymbol{\mathit{State}[C]}\\
\\
\boldsymbol{\mathit{State}[S]} &\stackrel{\mathit{def}}{=} (\mathit{call}[S.P],\top).\mathit{State}[S.P]'_1\\
&\;\;+\;\; (\mathit{call}[S.B]_1,\top).(\mathit{ret}[S.B]_1,0.01).\boldsymbol{\mathit{State}[S]}\\
&\;\;+\;\; (\mathit{call}[S.B]_2,\top).(\mathit{ret}[S.B]_2,0.01).\boldsymbol{\mathit{State}[S]}\\
&\;\;+\;\; (\mathit{call}[S.B]_3,\top).(\mathit{ret}[S.B]_3,0.01).\boldsymbol{\mathit{State}[S]}\\
\mathit{State}[S.P]'_1 &\stackrel{\mathit{def}}{=} (\mathit{ret}[S.P]_1,0.00203).\boldsymbol{\mathit{State}[S]}\\
&\;\;+\;\; (\mathit{ret}[S.P]_2,0.00795).\boldsymbol{\mathit{State}[S]}\\
&\;\;+\;\; (\mathit{ret}[S.P]_3,0.00002).\boldsymbol{\mathit{State}[S]}
\end{aligned}
$$

Figure 4.10: Object-level sequential processes for the client-server example

The $User_{C.B}$ component is defined using Equation 4.9 as follows:

$$\begin{array}{rcl} User_{C.B} & \overset{def}{=} & (call[C.B], 1).User'_{C.B} \\ User'_{C.B} & \overset{def}{=} & (ret[C.B]_1, \top).Nil + (ret[C.B]_2, \top).Nil \\ & + & (ret[C.B]_3, \top).Nil + (ret[C.B]_4, \top).Nil \end{array}$$

4.6 Model-Level Transformations

We have so far seen how to construct a PEPA model of a Siril program that has a single point of instantiation. Whilst this can allow us to answer some interesting questions relating to properties such as response time, we might also want to analyse the model in different contexts or environments. In this section, we will show how to do this by performing some transformations at the level of the PEPA model, without needing to modify our abstract interpretation and collecting semantics.

There are many ways in which we could transform a PEPA model of a Siril program, but we will describe just three of them here for illustrative purposes:

1. To model the case when there are multiple instances of a resource — for example, multiple clients or multiple servers — we can simply increase the number of copies of the component corresponding to that resource in the system equation. If we let c_i be the number of copies of object O_i, and c_U the number of users, then a more general version of Equation 4.10 is:

   $$User_{O_i.f_j}[c_U] \underset{*}{\bowtie} State(O_1)[c_1] \underset{*}{\bowtie} \cdots \underset{*}{\bowtie} State(O_M)[c_M]$$

   Here, we use the PEPA shorthand notation for aggregation, where $C[n]$ is defined (for $n > 0$):

   $$\begin{array}{rcl} C[1] & = & C \\ C[n] & = & C \parallel C[n-1] \end{array}$$

2. We have assumed so far that our program has a single instantiation. That is to say, there is a single user that interacts with the program by a single call to one of its methods. It is quite straightforward, however, to extend our approach to deal with multiple different instantiations. To do so, we perform a *separate* abstract interpretation for each instantiation, then generate the projected transitions for each of them in the usual way. The only difference is that we need to take the *union* of the sets of projected transitions when constructing the PEPA model. This ensures that the sequential PEPA components include the behaviour of every instantiation. We can then construct the system equation and user process(es)

to give the behaviour we desire. For example, if we want to model a system where multiple clients interact with a single server under different instantiations, we could write a system equation of the form:

$$\left(User_{Client.f_1} \parallel \ldots \parallel User_{Client.f_n}\right) \underset{*}{\bowtie} State(Client)[n] \underset{*}{\bowtie} State(Server)$$

3. The definition of the $User_{O_i.f_j}$ component given in Equation 4.9 leads to an absorbing state, which means that the model describes the transient behaviour of a single instantiation of the system. We might instead be interested in the *long run* behaviour, when many calls are made repeatedly by the user. We can model this by a simple modification to the $User_{O_i.f_j}$ component — making it cyclic by replacing the absorbing state *Nil* with a return to the state $User_{O_i.f_j}$:

$$\begin{array}{rcl} User_{O_i.f_j} & \overset{def}{=} & (Act(\langle \pi_{X'}(\mu_I)\rangle), r_{init}).User'_{O_i.f_j} \\ User'_{O_i.f_j} & \overset{def}{=} & \sum_{\mu \in TM}(Act((\pi''_X(\mu))), \top).User_{O_i.f_j} \end{array}$$

From this, we can solve the underlying CTMC of the model to obtain steady state measures such as throughput and utilisation.

All three of the above transformations are sound in the context of Siril, because objects are immutable, and do not interfere with one another — they represent independent resources. This means that creating multiple instances of a resource is reasonable, since this corresponds to having multiple resources in the system, such as multiple servers. When there are many users of the system, these also do not interfere with one another, except to contend for shared resources. This corresponds to a strict notion of contention, where only one user can interact with a resource at any one time. Note that because the objects are immutable, no race conditions are introduced in doing this. Furthermore, we cannot model a class-based language, where different instances of a class have different states, since we cannot create or destroy objects.

We will now consider some examples of applying these transformations. As an example of the third approach, let us examine our client-server example in the context of the cyclic modification to the $User_{C.B}$ process:

$$\begin{array}{rcl} User_{C.B} & \overset{def}{=} & (call[C.B], 1).User'_{C.B} \\ User'_{C.B} & \overset{def}{=} & (ret[C.B]_1, \top).User_{C.B} + (ret[C.B]_2, \top).User_{C.B} \\ & + & (ret[C.B]_3, \top).User_{C.B} + (ret[C.B]_4, \top).User_{C.B} \end{array}$$

Further to this, we can use the first and second approaches to construct a model with multiple clients, but only one server, and with multiple instantiations. A system equation for such a scenario with n clients, with each client instantiating `Client.buy`, is

Number of Clients (n)	Server Utilisation
1	0.44028
2	0.62057
3	0.71637
4	0.77496

Figure 4.11: Utilisation of the server in the presence of multiple clients

as follows:

$$User_{C.B}[n] \underset{*}{\bowtie} \text{“}State[C]\text{”}[n] \underset{*}{\bowtie} \text{“}State[S\,]\text{”}$$

We have placed $State[C]$ and $State[S\,]$ in quotation marks to avoid confusion with the notation for PEPA aggregation.

Now that we have a PEPA model, we can carry out various performance analyses using the available tools [114, 48, 178]. We will not go into this in great detail here, as we will discuss tool support for model checking PEPA models in some detail in Chapter 7. Figure 4.11, however, illustrates a simple application of the steady state solution of the model, showing the utilisation of the server with different numbers of clients — this is the proportion of time that the server is not in the state $State[S\,]$.

4.7 Abstract Interpretation as an MDP

The approach we have taken throughout Sections 4.4 to 4.6 has been aimed at constructing a PEPA model that represents the behaviour of a Siril program. Whilst PEPA is very useful as a performance modelling formalism, since it allows us to describe a CTMC in a compositional manner, it is not the only language we might wish to map to. In Section 4.4, we assigned a probability $p^\sharp(\mu^\sharp \vdash s \longrightarrow \mu'^\sharp \vdash s')$ to each transition between states $\mu^\sharp \vdash s$ and $\mu'^\sharp \vdash s'$ in the abstract interpretation of a method (see Equation 4.2). These probabilities are *approximate*, however, in that we effectively re-normalise the distribution after each transition.

To avoid this approximation, we could instead record an upper and lower *bound* on the probability of each transition: $[p^\sharp_L, p^\sharp_U]$. This explicitly introduces non-determinism, since we do not know the precise probability of the transition taking place — only that it lies within these bounds. In this section, we will first describe how to compute such bounds in the abstract interpretation, then give a brief discussion on how we might extend our collecting semantics in this situation.

To begin with, let us consider what happens if we only have an upper bound for the probability of each transition out of a state. It is possible to infer some information about the lower bounding probabilities simply by eliminating those distributions that are impossible (i.e. do not sum to one). This process is known as *delimiting* [63], and is illustrated in the following table, in a scenario with three possible successor states:

Next state	$\mu_1'^\sharp \vdash s_1'$	$\mu_2'^\sharp \vdash s_2'$	$\mu_2'^\sharp \vdash s_3'$
Upper-bounding probability	0.3	0.6	0.3
Bounding probability	$[0, 0.3]$	$[0, 0.6]$	$[0, 0.3]$
Delimited bounding probability	$[0.1, 0.3]$	$[0.4, 0.6]$	$[0.1, 0.3]$

Taking the transition to $\mu_2'^\sharp \vdash s_2'$ as an example, since 0.6 is an upper bound of the transition probability, it must lie in the interval $[0, 0.6]$. However, the maximum probability of *not* taking this transition is $0.3 + 0.3 = 0.6$, so we can refine the interval to $[0.4, 0.6]$. If we have lower bounds on the transition probabilities, we can use delimiting to compute upper bounds in the same way.

Let us now inductively calculate bounds for the transition probabilities in our abstract interpretation, starting with transitions out of the initial state $\mu_I^\sharp \vdash \bigcirc_{O.f}$. Since $\mu_I^\sharp(\mathbb{R}^N) = 1$, the upper-bounding probabilities from the initial state are given by:

$$p_U^\sharp(\mu_I^\sharp \vdash \bigcirc_{O.f} \to \mu'^\sharp \vdash s') = \frac{\mu'^\sharp(\mathbb{R}^N)}{\mu_I^\sharp(\mathbb{R}^N)} = \mu'^\sharp(\mathbb{R}^N)$$

The lower bounds can be calculated by delimiting.

For the probability of taking subsequent transitions, we can determine a *lower bound* as follows, given that the state $\mu^\sharp \vdash s$ is reached by a series of transitions of the form $\mu_I^\sharp \vdash \bigcirc_{O.f} \xrightarrow{[p_{1,L}^\sharp, p_{1,U}^\sharp]} \cdots \xrightarrow{[p_{n,L}^\sharp, p_{n,U}^\sharp]} \mu^\sharp \vdash s$. Recall that since the abstract interpretation is a tree, there is only one such path to each state:

$$p_L^\sharp(\mu^\sharp \vdash s \to \mu'^\sharp \vdash s') = \frac{1}{\mu^\sharp(\mathbb{R}^N)} \left(p_{1,L}^\sharp p_{2,L}^\sharp \cdots p_{n,L}^\sharp \left(1 - \frac{\sum_{\{\mu''^\sharp \neq \mu'^\sharp \mid \exists s''.\ \mu^\sharp \vdash s \to \mu''^\sharp \vdash s''\}} \mu'^\sharp(\mathbb{R}^N)}{\mu^\sharp(\mathbb{R}^N)} \right) \right)$$

We compute the upper bounds by delimiting. Intuitively, the above is the lower-bounding probability of reaching the state $\mu'^\sharp \vdash s'$ from the initial state, divided by the upper-bounding probability of reaching state $\mu^\sharp \vdash s$ from the initial state. The reason for computing the lower bound directly, and then using delimiting to obtain the upper bound, rather than vice versa, is to avoid potential numerical problems that could occur if we divide by a lower bound of zero.

This approach results in an abstract Markov chain (a type of Markov Decision Process), which contains non-determinism due to there being many possible distributions that satisfy the constraints on the transition probabilities out of a state. We could build a collecting semantics on top of this with only minor modifications to the technique of Section 4.5, mapping to a suitable process algebra such as Interactive Markov Chains (IMC) [87]. In IMC, rates and actions are separate prefixes, rather than being combined to form activities as in PEPA. If multiple actions are enabled, the choice between them is made non-deterministically. We could therefore use this to model the non-determinism between the upper and lower probability bounds on the transitions in the abstract interpretation.

On the basis of this discussion, it is important to realise that our choice of PEPA is a design decision, since it is by no means the only reasonable choice. Note, however, that abstract Markov chains are useful not only for modelling systems with inherent non-determinism, but for *abstracting* purely stochastic models. This is an idea that we will examine in some detail in the next two chapters — first in the context of Markov chains, and then in the context of PEPA.

Chapter 5

Stochastic Abstraction of Markov Chains

So far in this thesis, we have considered how to produce a Markovian model of a program by abstracting its source code. Although this abstraction is feasible for the class of programs we have considered, it will in general lead to models that are very large. Despite this, we need to be able to analyse them if we are to obtain performance information about the system. In this chapter and the next, we will explore the issues surrounding the analysis of large Markovian models, and in particular those expressed compositionally in the stochastic process algebra PEPA.

There are a number of different approaches to analysing Markov chains, which we can classify as belonging to a spectrum. At one end of this spectrum are *empirical* approaches, such as *simulation*, and at the other end are *analytical* approaches[1], such as stochastic *model checking*. In between, there are approaches such as statistical model checking, which use simulation to verify properties up to a certain confidence interval. In fact, a similar spectrum is found in techniques for qualitative verification of software, where we also have empirical approaches (testing) and analytical approaches (formal verification).

There are advantages and disadvantages to both types of approach. Simulating a Markov chain has the advantage that we can analyse large models, but it is expensive to compute, since we need a sufficiently large sample to obtain accurate results. Stochastic model checking is more efficient in this respect, and also more accurate, since we can say for certain whether or not a property is satisfied. However, since we often need

[1] Here, we use ‘analytical’ to mean the systematic study of the entire state space of the Markov chain, and not necessarily deriving a closed form expression for the solution of the chain.

to explore the entire reachable state space, there are severe limits on the size of models that we can analyse.

The state space explosion problem is a major issue for Markovian modelling in general, and we spent some time discussing the background to this in Section 2.5.3. This is particularly important given that we are concerned with highly parallel systems. Recall that if we have a model that contains just six copies of a component with ten states, its Markov chain could have as many as a million states. Because of this we need techniques for reducing the size — namely, *abstracting* — such models, so that they become small enough to analyse.

In Section 2.5.3 we discussed a number of approaches to tackling the state space explosion problem of Markov chains, but in this chapter we will focus on *state-based abstraction*. This involves combining, or aggregating, states in order to reduce the size of the model. The problem with this approach is that in most cases we cannot aggregate states and still end up with a Markov chain. Either we can try to construct a Markov chain that approximates the original as best as possible, or we can look for one that *bounds* the properties that we are interested in. We will consider two approaches to this — *abstract Markov chains* and *stochastic bounds* — in Section 5.6.

When we use a compositional modelling formalism such as PEPA, we can easily and compactly describe models that have extremely large state spaces. However, analysing such a model still requires its state space to be explored, which involves constructing the underlying Markov chain. Whilst it is then possible to abstract this Markov chain before analysing it, we run the risk that it may be too large to even represent. It would therefore be advantageous to perform the abstraction *compositionally* — at the level of the PEPA model — so that we can deal with much larger models. This will be the subject of the next chapter, but we first need to look at the existing techniques for state-based abstraction of Markov chains.

Many interesting properties of Markov chains can be verified using *stochastic model checking*. The basic problem is to determine whether a property p holds for some state s in a model $\mathcal{M}$ — namely, that:

$$\mathcal{M}, s \vDash p$$

Properties are typically expressed in a temporal logic, which allows us to reason about not only the current state, but the future behaviour of the model. When the model $\mathcal{M}$ is a Markov chain, properties p are usually expressed in a logic such as Probabilistic Computation Tree Logic (PCTL) [84] and PCTL* [15] (for discrete time), or Continuous Stochastic Logic (CSL) [16] (for continuous time).

In this chapter, we will begin by formally introducing discrete and continuous time Markov chains, in Section 5.1, so as to establish the notation that will be used in this chapter and the next. In Section 5.2 we will then present the logic CSL. Following this, we will formally introduce the notion of state-based abstraction in Section 5.3, and describe exact and approximate abstractions in Sections 5.4 and 5.5 respectively. Finally, we will look at *bounding* abstractions in more detail — in particular, abstract Markov chains and stochastic bounds — in Section 5.6.

5.1 Markov Chains in Discrete and Continuous Time

Before we look at abstracting and model checking Markov chains, we need to introduce some basic concepts. Let us begin by recalling the definitions of discrete and continuous time Markov chains, from Chapter 2:

Definition 2.5.1. *A* Discrete Time Markov Chain (DTMC) *is a tuple* $(S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, L)$*, where* S *is a countable and non-empty set of states,* $\boldsymbol{\pi}^{(0)} : S \rightarrow [0,1]$ *is an initial probability distribution over the states,* $\boldsymbol{P} : S \times S \rightarrow [0,1]$ *is a function describing the probability of transitioning between two states, and* $L : S \times AP \rightarrow \{\mathtt{tt}, \mathtt{ff}\}$ *is a labelling function over a finite set of propositions* AP*.*

Definition 2.5.2. *A* Continuous Time Markov Chain (CTMC) *is a tuple* $(S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$*, where* S*,* $\boldsymbol{\pi}^{(0)}$*,* $\boldsymbol{P}$ *and* L *are defined as for a DTMC, and* $r : S \rightarrow \mathbb{R}_{\geq 0}$ *assigns an exit rate to each state. If* $r(s) = 0$ *then no transitions are possible from state* s*, and we require that* $\boldsymbol{P}(s,s) = 1$*, and* $\boldsymbol{P}(s,s') = 0$ *for all* $s' \neq s$*.*

A *path* σ in a DTMC is a (possibly infinite) sequence of states $s_0, s_1, \ldots \in S$, such that for all $i < |\sigma| - 1$, $\boldsymbol{P}(s_i, s_{i+1}) > 0$. We write $\sigma[i] = s_i$, and $Paths(s)$ to be the set of paths such that $\sigma[0] = s$. A path σ in a CTMC is a (possibly infinite) alternating sequence of states and durations $s_0, t_0, s_1, t_1, \ldots$, such that $s_i \in S$, $t_i \in \mathbb{R}_{>0}$, and for all $i < |\sigma| - 1$, $\boldsymbol{P}(s_i, s_{i+1}) > 0$ and $r(s_i) > 0$. As for a DTMC path, we write $\sigma[i] = s_i$, but we additionally define $\delta(\sigma, i) = t_i$ (the time spent in state s_i), and $\sigma@t = \sigma[i]$, where i is the smallest index such that $t < \sum_{j=0}^{i} t_j$.

Often, a CTMC is described in terms of an infinitesimal generator matrix $\boldsymbol{Q}$, where $\boldsymbol{Q}(s,s')$ (where $s \neq s'$) gives the rate of transitioning between states s and s. In order for a Markov chain to be conservative, the diagonal elements are defined as

$\boldsymbol{Q}(s,s) = -\sum_{s' \neq s} \boldsymbol{Q}(s,s')$. This matrix can be computed from the rate function r and the probability transition matrix $\boldsymbol{P}$ as follows, where $\boldsymbol{I}$ is the identity matrix:

$$\boldsymbol{Q} = r(\boldsymbol{P} - \boldsymbol{I}) = r\boldsymbol{P} - r\boldsymbol{I}$$

Here, we define the multiplication of a matrix $\boldsymbol{M}$ by a rate function r as follows:

$$(r\boldsymbol{M})(s,s') = r(s)\boldsymbol{M}(s,s') \tag{5.1}$$

Every CTMC conforming to Definition 2.5.2 has an *embedded DTMC*, which can be obtained by simply discarding its rate function:

Definition 5.1.1. *The* embedded DTMC *of a CTMC* $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$ *is defined as* $Embed(\mathcal{M}) = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, L)$.

This, however, alters the behaviour of the Markov chain by throwing away the relative timing information of its states. In particular, the steady-state solution of the embedded DTMC will in general be different to that of the original CTMC. We can avoid this problem if we first *uniformise* the CTMC.

Definition 5.1.2. *The* uniformisation *of a CTMC* $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$*, with uniformisation rate* $\lambda \geq \max_{s \in S} r(s)$ *is given by* $Unif_\lambda(\mathcal{M}) = (S, \boldsymbol{\pi}^{(0)}, \overline{\boldsymbol{P}}, \overline{r}, L)$*, where* $\overline{r}(s) = \lambda$ *for all* $s \in S$*, and:*

$$\overline{\boldsymbol{P}}(s,s') = \begin{cases} \frac{r(s)}{\lambda}\boldsymbol{P}(s,s') & \text{if } s \neq s' \\ 1 - \frac{r(s)}{\lambda}\sum_{s'' \neq s}\boldsymbol{P}(s,s'') & \text{otherwise} \end{cases}$$

Essentially, uniformisation adjusts the CTMC by inserting self-loops, so that the exit rate of every state is the same. This preserves *weak bisimilarity* [19], which intuitively means that we preserve the relative rates of transitioning to different states.

5.2 The Continuous Stochastic Logic

To describe properties of a Markov chain, it is useful to have a logic for expressing them. Since we are primarily interested in CTMCs, we will focus on Continuous Stochastic Logic (CSL) [16], which is the most widely used logic in this setting. CSL is a branching-time temporal logic, and is an extension of Computation Tree Logic (CTL) [62]. It allows us to talk about the *probability* of a state satisfying some temporal property, and the *time interval* in which the property must hold.

Formulae in CSL are classified into *state formulae* Φ, and *path formulae* φ. The former are properties of individual states in the Markov chain — for example, that the steady state probability is greater than a certain value. The latter are properties that hold of paths (sequences of states) through the chain — for example, that a state property holds until some condition is met.

State formulae Φ are defined as follows, for $\trianglelefteq \in \{\leq, \geq\}$, $a \in AP$ and $p \in [0,1]$:

$$\Phi ::= \texttt{tt} \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{S}_{\trianglelefteq p}(\Phi) \mid \mathcal{P}_{\trianglelefteq p}(\varphi)$$

Path formulae φ have the following syntax, where $I = [a,b]$ is a non-empty interval over the reals, such that $a,b \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, and $a \leq b$:

$$\varphi ::= X^I\ \Phi \mid \Phi\ \mathcal{U}^I\ \Phi$$

Aside from atomic propositions and the standard Boolean connectives, there are three interesting types of property that can be expressed in CSL:

- A *steady state property* — $\mathcal{S}_{\trianglelefteq p}(\Phi)$ is satisfied if the steady state probability of being in the set of states satisfying Φ is $\trianglelefteq p$.
- A *timed next property* — $\mathcal{P}_{\trianglelefteq_p}(X^I\Phi)$ is satisfied of a state s if the probability that we leave the state at time $t \in I$, and the next state satisfies Φ, is $\trianglelefteq p$.
- A *timed until property* — $\mathcal{P}_{\trianglelefteq_p}(\Phi_1\, \mathcal{U}^I \Phi_2)$ is satisfied of a state s if the probability that we reach a state that satisfies Φ_2 at a time $t \in I$, and we only pass through states that satisfy Φ_1 along the way, is $\trianglelefteq p$.

As an example of a CSL timed until property, consider the following, for the set of atomic propositions $AP = \{Error, Completed\}$:

$$\mathcal{P}_{\geq 0.9}(\neg Error\ \mathcal{U}^{[0,10]}\ Completed)$$

This will be satisfied by all states from which there is a probability of at least 0.9 that we will reach a '*Completed*' state within 10 time units, without encountering any '*Error*' states before that point. Note that the unit of time is implicit to the model, and is only relevant with respect to our interpretation of the results.

An extension to the basic CSL syntax, used for example in the model checker PRISM [114], allows us to directly query the value of a steady state or path probability:

$$\Phi_T ::= \mathcal{S}_{=?}(\Phi) \mid \mathcal{P}_{=?}(\varphi)$$

$$
\begin{array}{lcll}
s & \models & \mathtt{tt} & \text{iff} \quad \text{true} \\
s & \models & a & \text{iff} \quad L(s,a) \\
s & \models & \Phi_1 \wedge \Phi_2 & \text{iff} \quad s \models \Phi_1 \text{ and } s \models \Phi_2 \\
s & \models & \neg\Phi & \text{iff} \quad s \not\models \Phi \\
s & \models & \mathcal{S}_{\trianglelefteq p}(\Phi) & \text{iff} \quad Prob_{\mathcal{S}}(s,\Phi) \trianglelefteq p \\
s & \models & \mathcal{P}_{\trianglelefteq p}(\varphi) & \text{iff} \quad Prob_{\mathcal{P}}(s,\varphi) \trianglelefteq p \\
\hline
\sigma & \models & \mathcal{X}^{I}\,\Phi & \text{iff} \quad \sigma[1] \models \Phi \text{ and } \delta(\sigma,0) \in I \\
\sigma & \models & \Phi_1\,\mathcal{U}^{I}\,\Phi_2 & \text{iff} \quad \exists t \in I.\ \sigma@t \models \Phi_2 \text{ and } \forall t' < t.\ \sigma@t' \models \Phi_1
\end{array}
$$

Figure 5.1: CSL semantics over $\mathcal{M} = (S, \boldsymbol{P}, \lambda, L)$

These are not state formulae in themselves, since they do not evaluate to a truth value, but evaluate to the *probability* of the property holding of a state. This does not affect the expressivity of CSL, but is of practical convenience for users of a model checker. The semantics of CSL is shown in Figure 5.1, over a CTMC $\mathcal{M} = (S, \boldsymbol{P}, \lambda, L)$. In defining this, we make use of the following measures:

$$
\begin{array}{rcl}
Prob_{\mathcal{S}}(s,\Phi) & = & \lim\limits_{t\to\infty} \Pr\{\sigma \in Paths(s) \mid \sigma@t \models \Phi\} \\
Prob_{\mathcal{P}}(s,\varphi) & = & \Pr\{\sigma \in Paths(s) \mid \sigma \models \varphi\}
\end{array}
$$

Formally, the above probability measure is defined as a Borel measure over sets of paths — we will not go into technical details here, but instead refer the reader to Section 2.3 of [17]. Importantly, it can be proven that the sets of paths in the above are indeed measurable. In [17], a full model checking algorithm for CSL is also presented.

In the remainder of this chapter, we will look at techniques for abstracting CTMCs, and many of these rely on the use of *uniformisation*. The uniformisation of a CTMC is weakly bisimilar to the original CTMC, but it is *not* the case that weak bisimilarity preserves all CSL properties. The problem is the timed next operator, and in fact all properties in CSL$\backslash X$ — the subset of CSL without the next operator — *are* preserved by weak bisimulation [19]. As a consequence, we will mostly consider just CSL$\backslash X$ from here on.

5.3 Abstraction of Markov Chains

Due to the complexity of the systems that we are usually interested in modelling, it is very common for Markov chains to be too large to analyse directly. In these circumstances the generator matrix is too large to represent and then solve, and in the worst case the state space itself may be too large to store. For such a Markov chain, if we want to apply model checking rather than simulation, then we are forced to take one (or both) of two fundamental approaches:

1. *Structural decomposition* of the Markov chain into components, such that a property of the original can be expressed in terms of properties of the components. The idea is one of divide and conquer — we can derive a property of the original Markov chain just by looking at its components, which are exponentially smaller in size.

2. *Abstraction* of the Markov chain, such that properties of the abstract chain are related to those of the original (we will clarify what we mean by this shortly). The key idea is to aggregate states of the Markov chain — in other words, we choose a partitioning of the state space, and do not distinguish between states in the same partition.

We use the word 'property' here in a general sense, since there are many different analyses we may wish to carry out on a Markov chain. In our case, we are interested in model checking CSL formulae on a CTMC, but we may alternatively be interested in just the steady state solution of the Markov chain, or a transient property such as the first passage time distribution between two states. In fact, as we will see later, we analyse steady state properties in a different way to other CSL formulae, due to the nature of the abstractions that we use.

There are a wide variety of structural decomposition techniques, which we gave an overview of in Section 2.5.3. We will focus in this chapter on state-based abstractions of Markov chains, primarily because the techniques can be applied to any Markov chain, and not just those of a particular structural form. The relationship between properties of the abstract and original Markov chain depends on the form of the abstraction, which may be *exact*, *approximate* or *bounding*, as we shall see shortly. First, however, we need to define precisely what we mean by an abstraction of a Markov chain.

Consider a Markov chain with a state space S. The basic idea of state space abstraction is to reduce S to a smaller abstract state space $S^\sharp$. To define this an abstraction, we need a mapping between the concrete and abstract states:

Definition 5.3.1. *An* abstraction *of a state space S is a pair $(S^\sharp, \alpha)$, where $\alpha : S \to S^\sharp$ is a surjective function that maps every concrete state to an abstract state. We define a corresponding* concretisation function*, $\gamma : S^\sharp \to \mathcal{P}(S)$, as $\gamma(s^\sharp) = \{ s \mid \alpha(s) = s^\sharp \}$.*

Since there are more concrete states than abstract, α defines an *aggregation* of the concrete states. As an aside, let us define a pair of functions $\alpha' : \mathcal{P}(S) \to \mathcal{P}(S^\sharp)$ and $\gamma' : \mathcal{P}(S^\sharp) \to \mathcal{P}(S)$, such that $\alpha'(S') = \bigcup_{s \in S'} \{\alpha(s)\}$ and $\gamma'(S'^\sharp) = \bigcup_{s^\sharp \in S'^\sharp} \gamma(s^\sharp)$. It is straightforward to show that α' and γ' form a *Galois connection*, which illustrates how this connects to the ideas of abstract interpretation from Chapter 4.

Consider two Markov chains, $\mathcal{M}$ and $\mathcal{M}^\sharp$, with state spaces S and $S^\sharp$ respectively, such that $(S^\sharp, \alpha)$ is an abstraction of S. Then, considering the possible properties of $\mathcal{M}$ and $\mathcal{M}^\sharp$ to be the set of all CSL formulae given a fixed set AP of atomic propositions, then we say that the abstraction is *exact* if for all states $s \in S$ and all properties p:

$$\mathcal{M}, s \models p \Leftrightarrow \mathcal{M}^\sharp, \alpha(s) \models p \tag{5.2}$$

This is a very strict requirement, and we will consider precisely what it means in Section 5.4. Any property that holds of a concrete state s must also hold of its corresponding abstract state $\alpha(s)$, and vice versa, which means that for the abstraction to be exact we must only be able to distinguish between two states s and s' if $\alpha(s) \neq \alpha(s')$.

Usually, when given an abstraction $(S^\sharp, \alpha)$ of a Markov chain $\mathcal{M}$, it will not be possible to construct an exact abstraction $\mathcal{M}^\sharp$. As an alternative, we could construct an *approximate abstraction*, where Equation 5.2 holds for as many properties as possible, rather than for all p. Unfortunately, it is difficult in general to reason about the error in such approximate abstractions — namely, knowing in advance which properties Equation 5.2 holds for. We will discuss this in more detail in Section 5.5.

If we want to build an abstraction so that we can reason about the error introduced, a better approach is to construct a *bounding abstraction* — all properties that hold of the abstraction should be guaranteed to hold of the original Markov chain, but not necessarily vice versa. More formally, for all properties p and all states $s \in S$:

$$\mathcal{M}, s \models p \Leftarrow \mathcal{M}^\sharp, \alpha(s) \models p \tag{5.3}$$

Note that if $\mathcal{M}^\sharp, \alpha(s) \not\models p$ and $\mathcal{M}^\sharp, \alpha(s) \not\models \neg p$, then we cannot say whether or not p holds of state s in the concrete Markov chain. When constructing a bounding abstraction, the challenge is to have as few properties that are ‘unknown’ as possible.

As an example, consider the CSL formulae $\mathcal{S}_{\geq 0.1}(\Phi)$ and $\mathcal{S}_{\leq 0.2}(\Phi)$, where Φ is an atomic proposition and $s \models \Phi \Leftrightarrow \alpha(s) \models \Phi$. If these hold of the abstraction $\mathcal{M}^\sharp$, then we know that the steady state probability of being in a state satisfying Φ is between 0.1 and 0.2. However, even if the actual steady state probability is 0.12, it may not be the case that $\mathcal{S}_{\leq 0.15}(\Phi)$ holds of the abstraction.

This illustrates the general principle of a bounding abstraction — that the abstract properties are *safe approximations*. If a property holds of the abstraction then we can be certain that it holds of the original, but if not then we cannot infer anything. For instance, if our property is a probability, then in general our abstraction will give a looser interval around this probability. It is important to note that the abstraction $\mathcal{M}^\sharp$ is not necessarily a Markov chain, and will in general be a *Markov Decision Process* (MDP) [146]. In Section 5.6, we will introduce two different bounding abstractions — *abstract Markov chains* and *stochastic bounds*.

5.4 Exact Abstraction of Markov Chains

Recall that an abstraction $(S^\sharp, \alpha)$ is exact for Markov chains $\mathcal{M}$ and $\mathcal{M}^\sharp$, if for all properties p and all states $s \in S$:

$$\mathcal{M}, s \vDash p \Leftrightarrow \mathcal{M}^\sharp, \alpha(s) \vDash p$$

Intuitively, since this means that we cannot distinguish between states that map to the same abstract state, it requires that the rate of transition between two abstract states must be independent of the particular concrete state we are in. In practical terms, this means that the memoryless property must still hold of the abstraction, and therefore it must still be a Markov chain. This condition is called *ordinary lumpability* [108]:

Definition 5.4.1. *An* ordinary lumping *of a DTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, L)$ is an abstraction $(S^\sharp, \alpha)$ such that for all states $s, s' \in S$, if $\alpha(s) = \alpha(s')$ then for all states $s^\sharp \in S^\sharp$:*

$$\sum_{t \in \gamma(s^\sharp)} \boldsymbol{P}(s,t) = \sum_{t \in \gamma(s^\sharp)} \boldsymbol{P}(s',t)$$

Definition 5.4.2. *An* ordinary lumping *of a CTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$ is an abstraction $(S^\sharp, \alpha)$ such that for all states $s, s' \in S$, if $\alpha(s) = \alpha(s')$ then for all states $s^\sharp \in S^\sharp$:*

$$\sum_{t \in \gamma(s^\sharp)} r(s)\boldsymbol{P}(s,t) = \sum_{t \in \gamma(s^\sharp)} r(s')\boldsymbol{P}(s',t)$$

If $(S^\sharp, \alpha)$ is an ordinary lumping, then it *induces* a new DTMC or CTMC over the state space $S^\sharp$, since it allows us to completely define the probabilities and transition rates between abstract states.

As an example, consider the Markov chain in Figure 5.2. Let us interpret it as a uniformised CTMC, where all states have an exit rate of 1 (i.e. $r(s) = 1$ for all s),

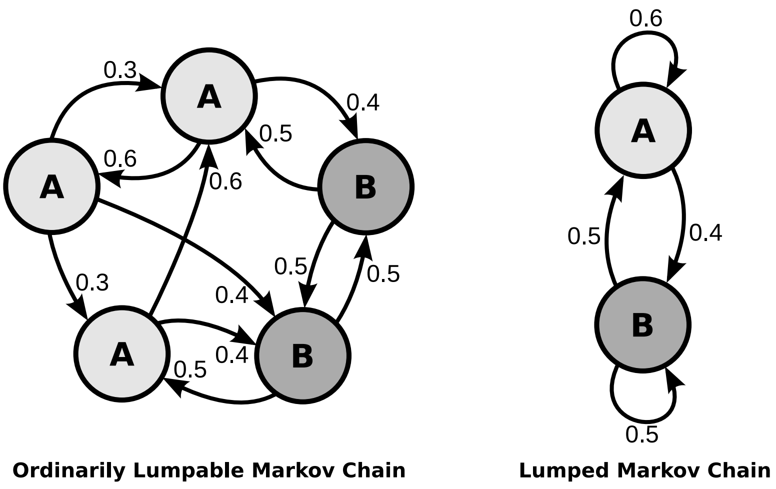


Figure 5.2: Ordinary lumpability of a Markov chain

and the transition probabilities are as given. For all states labelled A, the probability of moving to a state labelled B is 0.4, and that of moving to a state labelled A is 0.6. Similarly, every B state has the same probability distribution over moving to an A or a B state (0.5 in both cases). Since this satisfies the condition for ordinary lumpability, we can aggregate the CTMC to one that has only two states: A and B.

If we solve this CTMC, then the steady state probability of being in an abstract state (A or B) will be equal to the sum of the probabilities of being in each of its constituent states. In other words, solving the aggregated CTMC is equivalent to aggregating the solution of the original CTMC.

Ordinary lumpability ensures that an aggregated stochastic process is Markovian for all initial distributions over the state space. If this condition is relaxed so that only *some* initial distributions induce a Markov chain, then we have the notion of *weak lumpability* [108]. A special case of weak lumpability is *exact lumpability* [158], where the rate of transition *into* each state in the same partition is the same. More formally:

Definition 5.4.3. *An* exact lumping *of a DTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, L)$ is an abstraction $(S^\sharp, \alpha)$ such that for all states $s, s' \in S$, if $\alpha(s) = \alpha(s')$ then for all states $s^\sharp \in S^\sharp$:*

$$\sum_{t \in \gamma(s^\sharp)} \boldsymbol{P}(t,s) = \sum_{t \in \gamma(s^\sharp)} \boldsymbol{P}(t,s')$$

Definition 5.4.4. *An* exact lumping *of a CTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$ is an abstraction $(S^\sharp, \alpha)$ such that for all states $s, s' \in S$, if $\alpha(s) = \alpha(s')$ then for all states $s^\sharp \in S^\sharp$:*

$$\sum_{t \in \gamma(s^\sharp)} r(t)\boldsymbol{P}(t,s) = \sum_{t \in \gamma(s^\sharp)} r(t)\boldsymbol{P}(t,s')$$

This notion of lumpability ensures that the relative probabilities of being in the states of a partition are both determined, and fixed throughout the evolution of the Markov chain. However, we must start in an initial state that satisfies these relative distributions, in order to satisfy the Markov property. If a Markov chain is both ordinarily and exactly lumpable, it is said to be *strictly lumpable* [158].

For the remainder of this chapter, we will only consider ordinary lumpability, and therefore refer to this as lumpability without qualification. A useful consequence of ordinary lumpability is that it preserves all CSL properties [16][2], assuming that all states in the same partition satisfy the same atomic propositions.

5.5 Approximate Abstraction of Markov Chains

The problem with ordinary lumpability is that it only arises routinely when a Markov chain has a particular structure — for example, if it describes a system containing multiple copies of the same component in parallel. In general, we cannot rely on our model exhibiting lumpability, but since we are usually interested in combining states that have similar behaviour, it is not unreasonable that we might be *close* to being lumpable. If we formalise this idea, we arrive at the notion of *quasi-lumpability* [72]:

Definition 5.5.1. *A DTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, L)$ is* quasi-lumpable *with respect to an abstraction $(S^\sharp, \alpha)$, if the transition matrix $\boldsymbol{P}$ can be written as $\boldsymbol{P} = \boldsymbol{P}^- + \boldsymbol{P}^\epsilon$ such that:*

1. *$\boldsymbol{P}^-$ is a component-wise lower bound of $\boldsymbol{P}$, and is non-negative.*
2. *$\mathcal{M}^- = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}^-, L)$ is ordinarily lumpable with respect to $(S^\sharp, \alpha)$.*
3. *No element in $\boldsymbol{P}^\epsilon$ is greater than some small value ϵ.*

Since $\boldsymbol{P}^-$ and $\boldsymbol{P}^\epsilon$ may not be unique, we choose the alternative such that $||\boldsymbol{P}^\epsilon||_\infty$ is minimised[3].

This definition can be extended to a CTMC, but is typically done so in terms of the infinitesimal generator matrix $\boldsymbol{Q}$, rather than the separate rate function and probabilistic transition matrix as in Definition 5.4.2.

[2] Theorem 3 of [16] shows that two F-bisimilar states in a CTMC satisfy the same set F of CSL properties, and the definition of F-bisimilarity coincides with ordinary lumping equivalence.

[3] The infinity norm $||\boldsymbol{P}||_\infty$ of a matrix $\boldsymbol{P}$ is defined to be the maximum absolute row sum of the matrix.

Quasi-lumpability can be used as approximate abstraction, if we make $\boldsymbol{P}^{-}$ into a stochastic matrix by adding probability mass to the diagonal elements — hence preserving lumpability. If we do this, however, it becomes very difficult to reason about the error that we introduce — for both steady state and transient measures. There are more intelligent approaches that can *bound* the steady state probabilities for quasi-lumpable Markov chains [72], but these are only useful for Markov chains that have a particular structure, such as being nearly-completely decomposable (NCD) [57]. We would therefore like to use abstractions that can be applied more generally, which leads us to the techniques in the next section.

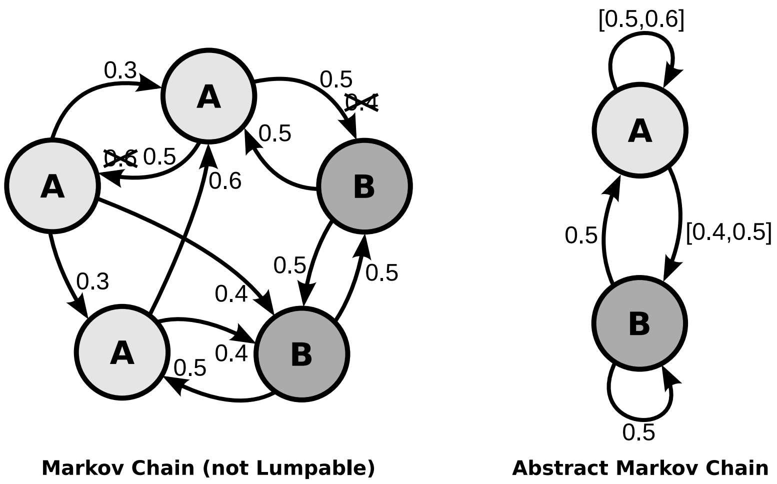
5.6 Bounding Abstraction of Markov Chains

As we have seen, most abstractions of a Markov chain are not ordinarily lumpable, so to avoid using approximate methods we need to find a way to safely *bound* the properties that we are interested in. To this end, we will consider two approaches to constructing a bounding abstraction of a Markov chain:

1. Determine the maximum and minimum possible transition rates between the abstract states. This defines not a Markov chain, but a *set* of Markov chains, on which we can perform model checking using a three-valued version of CSL (i.e. it has truth values of true (`tt`), false (`ff`) and maybe (?)). This is the approach of *abstract* — or *interval* — *Markov chains* [63, 102]. Note that there are also similar approaches using Markov decision processes, where we make a non-deterministic choice between which concrete state an abstract state is in, rather than having bounds on the rate of each transition [56].

2. Modify the original CTMC by altering the rates so that the abstraction becomes lumpable. In order for this to be useful, we ensure that the modification yields an upper (or lower) bound to the property that we are interested in. This is the approach of *stochastic bounds* [174]. Note that there has been related work in producing purely probabilistic abstractions of Markov chains and Markov decision processes [43].

We will describe these approaches in Section 5.6.1 and Section 5.6.2 respectively. In particular, abstract Markov chains are useful for model checking the transient (path) properties of CSL, while stochastic bounds can be used for the analysis of monotone properties, such as the steady state distribution.



Markov Chain (not Lumpable)

Abstract Markov Chain

Figure 5.3: A non-lumpable abstraction of a Markov chain

5.6.1 Abstract Markov Chains

Given a CTMC $\mathcal{M}$ and an abstraction $(S^\sharp, \alpha)$, we have seen that this does not give rise to a Markov chain in general. This is because the probability of making a transition between two abstract states (or the rate of the transition, in the case of a CTMC) varies, depending on which concrete state we were in. As an example, consider Figure 5.3, in which we have slightly modified the transition probabilities of the lumpable Markov chain from Figure 5.2. Here, the probability of moving from an A state to a B state depends on which particular A state we are in, and can be either 0.4 or 0.5, hence we cannot construct an aggregate CTMC.

Instead, if we label the transitions by an *interval* of probabilities that are possible, this gives rise to an *abstract Markov chain*. The notion of an abstract DTMC was introduced in [102, 63], and extended to continuous time in [105] by means of uniformisation. The idea is closely related to Markov Decision Processes (MDPs) [146], in that the transitions have both a probabilistic and a non-deterministic component. An abstract CTMC is defined as follows (as per [105]):

Definition 5.6.1. *An* abstract CTMC *is a tuple* $(S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^L, \boldsymbol{P}^U, \lambda, L^\sharp)$*, where* $S^\sharp$ *is a finite non-empty set of states,* $\boldsymbol{\pi}^{(0)\sharp} : S^\sharp \to [0,1]$ *is the initial probability distribution over the states, and* $\boldsymbol{P}^L, \boldsymbol{P}^U : S^\sharp \times S^\sharp \to [0,1]$ *are sub-stochastic and super-stochastic matrices respectively, such that for all states* $s, s' \in S^\sharp$*,* $\boldsymbol{P}^L(s,s') \le \boldsymbol{P}^U(s,s')$*.* λ *is the uniformisation constant, denoting the exit rate for every state, and we have a labelling function* $L^\sharp : S^\sharp \times AP \to \{\mathtt{tt}, \mathtt{ff}, ?\}$*.*

Note that the labelling function contains a third truth assignment, '?', which signifies uncertainty (some of the concrete states satisfy the property, but some do not). The truth assignments naturally form a partial order relating to the information they provide: $\mathtt{ff} \sqsubset ?$ and $\mathtt{tt} \sqsubset ?$. The abstract Boolean operators $\neg^\sharp$ and $\wedge^\sharp$ are defined as:

$\neg^\sharp$	
tt	ff
ff	tt
?	?

$\wedge^\sharp$	tt	ff	?
tt	tt	ff	?
ff	ff	ff	ff
?	?	ff	?

Note that we could arrive at this by taking the truth values to be elements of $\mathcal{P}(\{\mathtt{tt},\mathtt{ff}\}) \setminus \{\,\}$, where ? corresponds to $\{\mathtt{tt},\mathtt{ff}\}$, $\sqsubset$ corresponds to $\subset$, and we define $\neg S = \{\neg s \mid s \in S\}$.

The semantics of three-valued CSL over an abstract CTMC is shown in Figure 5.4. This makes use of the following measures, which are analogues of those used in the two-valued semantics:

$$\begin{aligned}
Prob^\sharp_{\mathcal{S}}(s,\Phi) &= \lim_{t\to\infty}\Big[\Pr\{\sigma \in Paths(s) \mid [\![\Phi]\!](\sigma@t) = \mathtt{tt}\}, \\
&\qquad 1 - \Pr\{\sigma \in Paths(s) \mid [\![\Phi]\!](\sigma@t) = \mathtt{ff}\}\Big] \\
Prob^\sharp_{\mathcal{P}}(s,\varphi) &= \Big[\Pr\{\sigma \in Paths(s) \mid [\![\varphi]\!](\sigma) = \mathtt{tt}\}, \\
&\qquad 1 - \Pr\{\sigma \in Paths(s) \mid [\![\varphi]\!](\sigma) = \mathtt{ff}\}\Big] \\
Test_{\trianglelefteq p}([p_L, p_U]) &= \begin{cases} \mathtt{tt} & \text{if } p \trianglelefteq p_L \text{ and } p \trianglelefteq p_U \\ \mathtt{ff} & \text{if } \neg(p \trianglelefteq p_L) \text{ and } \neg(p \trianglelefteq p_U) \\ ? & \text{otherwise} \end{cases}
\end{aligned}$$

Our definition of an abstract CTMC induces a natural partial order.

Definition 5.6.2. *If $\mathcal{M}^\sharp_1 = (S^\sharp_1, \boldsymbol{\pi}^{(0)\sharp}_1, \boldsymbol{P}^L_1, \boldsymbol{P}^U_1, \lambda_1, L^\sharp_1)$ and $\mathcal{M}^\sharp_2 = (S^\sharp_2, \boldsymbol{\pi}^{(0)\sharp}_2, \boldsymbol{P}^L_2, \boldsymbol{P}^U_2, \lambda_2, L^\sharp_2)$, then we say that $\mathcal{M}^\sharp_1 \leq \mathcal{M}^\sharp_2$ if:*

1. *$S^\sharp_1 = S^\sharp_2$, $\boldsymbol{\pi}^{(0)\sharp}_1 = \boldsymbol{\pi}^{(0)\sharp}_2$, $\lambda_1 = \lambda_2$ and $L^\sharp_1 = L^\sharp_2$.*

2. *For all $s, s' \in S^\sharp_1$: $\boldsymbol{P}^L_2(s,s') \leq \boldsymbol{P}^L_1(s,s') \leq \boldsymbol{P}^U_1(s,s') \leq \boldsymbol{P}^U_2(s,s')$.*

Intuitively, $\mathcal{M}^\sharp_2$ is an over-approximation of $\mathcal{M}^\sharp_1$, since a greater range of transition probabilities are possible. Note that this is strictly coarser than a simulation ordering, since we can only compare two abstract CTMCs that have have the same state space, uniformisation constant, and initial distribution. It has the advantage, however, of being easy to compute, and is sufficient for our purposes since we only construct

$$
\begin{array}{lcl}
\llbracket \texttt{tt} \rrbracket(s) & = & \texttt{tt} \\
\llbracket a \rrbracket(s) & = & L^{\sharp}(s,a) \\
\llbracket \Phi_1 \wedge \Phi_2 \rrbracket(s) & = & \llbracket \Phi_1 \rrbracket(s) \wedge^{\sharp} \llbracket \Phi_2 \rrbracket(s) \\
\llbracket \neg \Phi \rrbracket(s) & = & \neg^{\sharp} \llbracket \Phi_1 \rrbracket(s) \\
\llbracket \mathcal{S}_{\trianglelefteq p}(\Phi) \rrbracket(s) & = & \mathit{Test}_{\trianglelefteq p}(\mathit{Prob}^{\sharp}_{\mathcal{S}}(s,\Phi)) \\
\llbracket \mathcal{P}_{\trianglelefteq p}(\varphi) \rrbracket(s) & = & \mathit{Test}_{\trianglelefteq p}(\mathit{Prob}^{\sharp}_{\mathcal{P}}(s,\varphi)) \\
\hline
\llbracket \mathcal{X}^{I} \Phi \rrbracket(\sigma) & = & \begin{cases} \llbracket \Phi \rrbracket(\sigma[1]) & \text{if } \delta(\sigma,0) \in I \\ \texttt{ff} & \text{otherwise} \end{cases} \\
\llbracket \Phi_1 \, \mathcal{U}^{I} \, \Phi_2 \rrbracket(\sigma) & = & \begin{cases} \texttt{tt} & \text{if } \exists t \in I.\ \llbracket \Phi_2 \rrbracket(\sigma @ t) = \texttt{tt} \\ & \qquad \text{and } \forall t' < t.\ \llbracket \Phi_1 \rrbracket(\sigma @ t') = \texttt{tt} \\ \texttt{ff} & \text{if } \forall t \in I.\ \llbracket \Phi_2 \rrbracket(\sigma @ t) = \texttt{ff} \\ & \qquad \text{or } \exists t' < t.\ \llbracket \Phi_1 \rrbracket(\sigma @ t') = \texttt{ff} \\ ? & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.4: Three-valued CSL semantics over $\mathcal{M}^{\sharp} = (S^{\sharp}, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^{L}, \boldsymbol{P}^{U}, \lambda, L^{\sharp})$

abstractions that satisfy the above conditions — i.e. we do not need to compare two arbitrary abstract CTMCs. Importantly, $\mathcal{M}^{\sharp}_1 \leq \mathcal{M}^{\sharp}_2$ implies that $\mathcal{M}^{\sharp}_2$ can simulate $\mathcal{M}^{\sharp}_1$.

If we have a uniform CTMC $\mathcal{M}$ and an abstraction $(S^{\sharp}, \alpha)$, then we can uniquely define an abstract CTMC (the closest abstraction) as follows:

Definition 5.6.3. *The abstract CTMC $\mathcal{M}^{\sharp} = \mathit{Abs}_{(S^{\sharp},\alpha)}(\mathcal{M})$ induced by an abstraction $(S^{\sharp}, \alpha)$ on a uniform CTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$ is defined as follows. Since $\mathcal{M}$ is uniformised, there is a constant λ such that $r(s) = \lambda$ for all $s \in S$:*

$$\mathit{Abs}_{(S^{\sharp},\alpha)}(\mathcal{M}) = (S^{\sharp}, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^{L}, \boldsymbol{P}^{U}, \lambda, L^{\sharp})$$

where:

$$
\begin{array}{lcl}
\boldsymbol{\pi}^{(0)\sharp}(s^{\sharp}) & = & \displaystyle\sum_{s \in \gamma(s^{\sharp})} \boldsymbol{\pi}^{(0)}(s) \\
\boldsymbol{P}^{L}(s^{\sharp}_1, s^{\sharp}_2) & = & \displaystyle\min_{s_1 \in \gamma(s^{\sharp}_1)} \sum_{s_2 \in \gamma(s^{\sharp}_2)} \boldsymbol{P}(s_1, s_2) \\
\boldsymbol{P}^{U}(s^{\sharp}_1, s^{\sharp}_2) & = & \displaystyle\max_{s_1 \in \gamma(s^{\sharp}_1)} \sum_{s_2 \in \gamma(s^{\sharp}_2)} \boldsymbol{P}(s_1, s_2) \\
L^{\sharp}(s^{\sharp}, a) & = & \begin{cases} \texttt{tt} & \text{if } \forall s \in \gamma(s^{\sharp}).\ L(s,a) = \texttt{tt} \\ \texttt{ff} & \text{if } \forall s \in \gamma(s^{\sharp}).\ L(s,a) = \texttt{ff} \\ ? & \text{otherwise} \end{cases}
\end{array}
$$

In [105], a semantics for CSL under the above three-valued logic is given, along with a model checking algorithm for CSL$\backslash X$, without the steady state operator. We need to exclude the timed next operator, because its validity is not preserved after uniformisation of the CTMC. An algorithm for model checking abstract Markov chains is given in [18, 105], and we have implemented this in our tool, which we describe in Chapter 7. However, since the focus of this thesis is on techniques for abstraction, rather than model checking techniques, we will not describe the algorithms here.

5.6.2 Stochastic Bounding of Markov Chains

An abstract Markov chain can be used as a bounding abstraction of a Markov chain, if we are interested in transient properties, such as a CSL path property. Since the transitions have *intervals* of probabilities, however, we cannot compute a steady state distribution for an abstract Markov chain. To bound steady state probabilities, there are two fundamental approaches we could take. The first is to look at algorithms for bounding long-run averages of an abstract Markov chain, in the style of De Alfaro [7]. The second, which we consider, is to construct upper- and lower-bounding *Markov chains*, such that their steady state distributions bound that of the original Markov chain. Importantly, by ensuring that these bounds are lumpable, we can reduce the size of the Markov chains to solve. To do this, we need a notion of *stochastic ordering* of probability distributions.

There exist a number of stochastic orderings, for which Stoyan's book [174] is a detailed reference, but for our purposes we will use only the *strong stochastic order*, which we will denote $\leq_{\mathrm{st}}$.

Definition 5.6.4. *Let X and Y be random variables on a partially ordered space $(S, \prec)$. We say that X is less than Y in the strong stochastic order, namely $X \leq_{\mathrm{st}} Y$, if for all non-decreasing functions f, $\mathbb{E}[f(X)] \leq \mathbb{E}[f(Y)]$.*

Note that $(S, \prec)$ can be any partial order, and we will see some specific examples in the next chapter. This definition is equivalent to saying that $\mathbf{Pr}(X \succ s) \leq \mathbf{Pr}(Y \succ s)$, for all $s \in S$. In particular, since we can describe a discrete random variable in terms of a vector representation of its probability distribution, a more practical definition is the following:

Definition 5.6.5. *Let X be a random variable over $(S, \prec)$, and $\boldsymbol{x}$ be a vector of length $|S|$, such that $\mathbf{Pr}(X \succ s) = \sum_{s' \succ s} \boldsymbol{x}(s')$. Let $\boldsymbol{y}$ be defined similarly for a random variable*

Y over $(S,\prec)$. We say that $\boldsymbol{x} \leq_{\mathrm{st}} \boldsymbol{y}$ if for all $s \in S$:

$$\sum_{s' \succ s} \boldsymbol{x}(s') \leq \sum_{s' \succ s} \boldsymbol{y}(s')$$

It follows that $\boldsymbol{x} \leq_{\mathrm{st}} \boldsymbol{y}$ implies that $X \leq_{\mathrm{st}} Y$, and so we will describe random variables in terms of probability vectors from here on.

The strong stochastic order extends naturally to Markov chains. The following definition applies in both the discrete and continuous time settings:

Definition 5.6.6. *Let $\{X_t\}$ and $\{Y_t\}$ be Markov chains over the partially ordered state space $(S,\prec)$. Then $\{X_t\} \leq_{\mathrm{st}} \{Y_t\}$ if for all t, $X_t \leq_{\mathrm{st}} Y_t$.*

This is the classical definition, where we consider a Markov chain to be a set of random variables, indexed by the time t — two Markov chains are comparable if their probability distributions over $(S,\prec)$ are comparable at all times t. This definition of a Markov chain, however, differs from that in Section 5.1, and so to be practical, we need to define the strong stochastic order in terms of the transition matrices of Markov chains.

To do this, we need to introduce two important properties of stochastic matrices: *comparability* and *monotonicity*. We will assume that the state space of a matrix $\boldsymbol{P}$ (i.e. its row and column indices) is a partially ordered set $(S_{\boldsymbol{P}},\prec_{\boldsymbol{P}})$, and we will omit the subscript when it is clear from context. Furthermore, we will use the notation $\boldsymbol{P}(i,*)$ to denote row i of matrix $\boldsymbol{P}$, which is itself a row vector.

Definition 5.6.7. *Two stochastic matrices, $\boldsymbol{P}$ and $\boldsymbol{P}'$, are* comparable *such that $\boldsymbol{P} \leq_{\mathrm{st}} \boldsymbol{P}'$, if they share the same state space $(S,\prec)$, and for all $s \in S$, $\boldsymbol{P}(s,*) \leq_{\mathrm{st}} \boldsymbol{P}'(s,*)$.*

Definition 5.6.8. *An $|S|\times|S|$ stochastic matrix $\boldsymbol{P}$ is* monotone *if for all row vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ of length $|S|$, $\boldsymbol{u} \leq_{\mathrm{st}} \boldsymbol{v}$ implies that $\boldsymbol{u}\boldsymbol{P} \leq_{\mathrm{st}} \boldsymbol{v}\boldsymbol{P}$. Equivalently, $\boldsymbol{P}$ is monotone if for all $s,s' \in S$, $s \prec s' \Rightarrow \boldsymbol{P}(s,*) \leq_{\mathrm{st}} \boldsymbol{P}(s',*)$.*

Note that comparability and monotonicity are quite different concepts. Intuitively, comparability is about comparing the same row in two different matrices, whereas monotonicity is about comparing two different rows in the same matrix.

Using these notions of stochastic comparison and monotonicity, we can now provide an alternative definition of the strong stochastic order on DTMCs:

Definition 5.6.9. *Consider the DTMCs $\mathcal{M}_1 = (S,\boldsymbol{\pi}_1^{(0)},\boldsymbol{P}_1,L)$ and $\mathcal{M}_2 = (S,\boldsymbol{\pi}_2^{(0)},\boldsymbol{P}_2,L)$. We say that $\mathcal{M}_1 \leq_{\mathrm{st}} \mathcal{M}_2$ if:*

1. $\boldsymbol{\pi}_1^{(0)} \leq_{\mathrm{st}} \boldsymbol{\pi}_2^{(0)}$.

2. $\boldsymbol{P}_1 \leq_{\mathrm{st}} \boldsymbol{P}_2$.

3. $\boldsymbol{P}_1$ *or* $\boldsymbol{P}_2$ *is monotone.*

To motivate why the third condition is required, let us consider a small example where two stochastic matrices are comparable, but neither is monotone. We will assume a totally ordered state space, given by the indices of the states in the matrix. If we start with the same initial distribution, then we quickly find a point in time where the distributions over the states are not comparable:

$$\boldsymbol{P}: \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \leq_{\mathrm{st}} \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \qquad \begin{array}{rccc} \boldsymbol{\pi}^{(0)}: & \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} & \leq_{\mathrm{st}} & \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \\ \boldsymbol{\pi}^{(1)}: & \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} & \leq_{\mathrm{st}} & \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \\ \boldsymbol{\pi}^{(2)}: & \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} & \not\leq_{\mathrm{st}} & \begin{bmatrix} \frac{1}{4} & 0 & \frac{3}{4} \end{bmatrix} \end{array}$$

Given this definition for stochastic comparason of DTMCs, we can also compare two CTMCs — by uniformising them and comparing their embedded DTMCs:

Definition 5.6.10. *Two CTMCs* $\mathcal{M}_1 = (S, \boldsymbol{\pi}_1^{(0)}, \boldsymbol{P}_1, r_1, L)$ *and* $\mathcal{M}_2 = (S, \boldsymbol{\pi}_2^{(0)}, \boldsymbol{P}_2, r_2, L)$ *are comparable such that* $\mathcal{M}_1 \leq_{\mathrm{st}} \mathcal{M}_2$ *if for all* $\lambda \geq \max(\max_{s \in S} r_1(s), \max_{s \in S} r_2(s))$*:*

$$\mathit{Embed}(\mathit{Unif}_\lambda(\mathcal{M}_1)) \leq_{\mathrm{st}} \mathit{Embed}(\mathit{Unif}_\lambda(\mathcal{M}_2))$$

Since stochastic comparison of CTMCs is defined in terms of stochastic comparison of DTMCs, we need only consider the latter, without loss of generality, for the remainder of this section. The need to uniformise CTMCs, however, will become an important consideration when we apply these techniques compositionally to PEPA models in Section 6.4.

For stochastic bounds to be of practical use, we need algorithms to construct monotone upper and lower-bounding matrices, given the probability transition matrix of a DTMC. Furthermore, not only do we need them to be bounding, but they must be *lumpable* with respect to the desired abstraction.

In [69], an algorithm is given that derives an irreducible and lumpable bounding DTMC from an initial DTMC, assuming that it has a totally-ordered state space. Recall that irreducibility means that the DTMC has a strongly connected state space. We will describe this in two stages — first dealing with the steps needed to ensure a monotone and irreducible upper bound, and secondly with how to ensure that this is lumpable (with respect to a partitioning of the states). The first stage begins with the algorithm

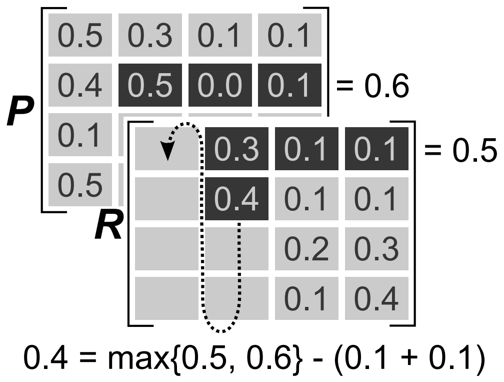


Figure 5.5: Algorithm for computing a monotone upper bound of a stochastic matrix

by Abu-Amsha and Vincent [5], which finds a monotone upper-bounding transition matrix for a DTMC. The idea is to observe that, for two stochastic matrices $\boldsymbol{P}$ and $\boldsymbol{R}$ of the same dimensions, the following inequalities must hold if $\boldsymbol{P} \leq_{\mathrm{st}} \boldsymbol{R}$ and $\boldsymbol{R}$ is monotone:

1. For all i, $\boldsymbol{P}(i,*) \leq_{\mathrm{st}} \boldsymbol{R}(i,*)$.
2. For all i, $\boldsymbol{R}(i,*) \leq_{\mathrm{st}} \boldsymbol{R}(i+1,*)$.

To arrive at the basic algorithm, we set the first row of $\boldsymbol{R}$ equal to that of $\boldsymbol{P}$ ($\boldsymbol{R}(1,*) = \boldsymbol{P}(1,*)$), and then set each subsequent row according to the maximum of the left-hand sides of the above inequalities. Since the ordering on the states is total, $\boldsymbol{P}(i,*) \leq_{\mathrm{st}} \boldsymbol{R}(i,*)$ means that $\sum_{k=j}^{n} \boldsymbol{P}(i,k) \leq \sum_{k=j}^{n} \boldsymbol{R}(i,k)$, for all j. This leads to the following definition of $\boldsymbol{R}$:

$$\boldsymbol{R}(i,j) = \max\left\{\sum_{k=j}^{n} \boldsymbol{R}(i-1,k), \sum_{k=j}^{n} \boldsymbol{P}(i,k)\right\} - \sum_{k=j+1}^{n} \boldsymbol{R}(i,k)$$

We can iteratively construct the matrix $\boldsymbol{R}$ based on the above equation, starting with the first row and the last column, and iterating down the rows before moving onto the next column. This is illustrated in Figure 5.5, which shows how we take the maximum of the sum of probabilities for the same row in the original matrix and the previous row in the upper-bounding matrix. Iterating in this order ensures that each sum is known when we need it.

To compute a lower-bounding matrix, we can use the following, setting the last row of $\boldsymbol{R}$ equal to that of $\boldsymbol{P}$ and reversing the order of iteration over the rows:

$$\boldsymbol{R}(i,j) = \min\left\{\sum_{k=0}^{j} \boldsymbol{R}(i+1,k), \sum_{k=0}^{j} \boldsymbol{P}(i,k)\right\} - \sum_{k=0}^{j-1} \boldsymbol{R}(i,k)$$

From here on, we will consider only the algorithm for the upper-bounding matrix, as that for the lower-bounding matrix is similar.

Unfortunately, this basic algorithm does not guarantee that if $\boldsymbol{P}$ is irreducible then $\boldsymbol{R}$ will also be, since it is possible to delete transitions. Fourneau *et al.* [69] address this by a slight modification to the algorithm, so that we avoid unnecessarily deleting transitions. In particular, if the basic algorithm gives $\boldsymbol{R}(i,j) = 0$, and we have not yet consumed all of the probability mass for that row ($\sum_{k=j+1}^{n} \boldsymbol{R}(i,k) < 1$), then in the case that $\boldsymbol{P}(i,j) > 0$ (i.e. we would delete a transition) we add a small value (ϵ — which is a parameter of the algorithm — times the remaining probability mass) to the element, which avoids deleting it. We do the same if these conditions hold when $i = j-1$ — i.e. we are to the right of a diagonal element, even if the original had zero probability. This avoids placing all of the probability mass on the diagonal, which would result in an absorbing state. The stochastic bounding algorithm, including this modification, is shown in Algorithm 1.

Algorithm 1 Computation of an upper-bounding, monotone and irreducible stochastic matrix (from [69])

```
for $j \leftarrow 1$ to $n$ do
    $\boldsymbol{R}(1,j) \leftarrow \boldsymbol{P}(1,j)$
end for
for $i \leftarrow 2$ to $n$ do
    $\boldsymbol{R}(i,n) \leftarrow \max\{\boldsymbol{R}(i-1,n), \boldsymbol{P}(i,n)\}$
end for
for $j \leftarrow n-1$ to $1$ do
    for $i \leftarrow 1$ to $n$ do
        $\boldsymbol{R}(i,j) \leftarrow \max\left\{0, \max\left\{\sum_{k=j}^{n} \boldsymbol{R}(i-1,k), \sum_{k=j}^{n} \boldsymbol{P}(i,k)\right\} - \sum_{k=j+1}^{n} \boldsymbol{R}(i,k)\right\}$
        if $\boldsymbol{R}(i,j) \leftarrow 0 \wedge \sum_{k=j+1}^{n} \boldsymbol{R}(i,k) < 1 \wedge (\boldsymbol{P}(i,j) > 0 \vee i = j-1)$ then
            $\boldsymbol{R}(i,j) \leftarrow \epsilon \times \left(1 - \sum_{k=j+1}^{n} \boldsymbol{R}(i,k)\right)$
        end if
    end for
end for
```

To produce an upper-bounding matrix that is not only monotone and irreducible, but describes a *lumpable* DTMC with respect to a given partitioning, the algorithm of [69] has a further step. It is assumed that all states in a given partition are adjacent to one another in the total ordering on the state space. This step, known as *normalisation*, ensures that all the row sums in the same partition are the same, by adding probability mass to the lowest-valued row in the state space ordering. This preserves monotonicity, and always results in a stochastic matrix. Algorithm 2 performs this normalisation for a given partition k, assuming that there are K partitions in total, and is called after filling in the group of columns corresponding to partition k, where $b(k)$ is the first state in partition k, and $e(k)$ is the last state.

Algorithm 2 Normalisation of a stochastic bounding matrix, to ensure lumpability (from [69])

```
for $y \leftarrow 1$ to $K$ do
    $sum \leftarrow \sum_{j=b(k)}^{e(k)} \boldsymbol{R}(e(y), j)$
    for $i \leftarrow b(y)$ to $e(y) - 1$ do
        $\boldsymbol{R}(i, b(k)) \leftarrow sum - \sum_{j=b(k)+1}^{e(k)} \boldsymbol{R}(i, j)$
    end for
end for
```

The basic properties of this stochastic bounding algorithm, given by Algorithms 1 and 2, are as follows, given a stochastic matrix $\boldsymbol{P}$ as input, and an output matrix $\boldsymbol{R}$ generated by the algorithm [69]:

1. If $\boldsymbol{P}$ is irreducible, $\boldsymbol{P}(1,1) > 0$, and every row of the lower triangle of $\boldsymbol{P}$ has at least one positive element, then $\boldsymbol{R}$ is irreducible.

2. $\boldsymbol{R}$ is monotone, and $\boldsymbol{P} \leq_{\text{st}} \boldsymbol{R}$, under the ordering of the indices of the matrices.

3. Given a partitioning of the state space, such that all the states in the same partition have adjacent indices, the DTMC described by $\boldsymbol{R}$ is lumpable with respect to this partitioning.

We can now state some basic complexity results for the algorithm [69]:

Property 5.6.11. *The worst case time complexity for Fourneau's algorithm is $O(n^2)$, where n is the size of the state space (i.e. $\boldsymbol{P}$ and $\boldsymbol{R}$ are $n \times n$ matrices). The space complexity is $O(n)$, in terms of the memory required in computing the bound (but not to store the result).*

Intuitively, this is because we can keep a running count of the row sums as we iterate through the matrix, to avoid having to re-compute them for every element of the matrix. The memory requirement is $O(n)$ because we need two vectors of length n to store these sums — for the current row of $\boldsymbol{P}$, and the previous row of $\boldsymbol{R}$.

The main disadvantage of this algorithm is that it only applies when the state space is totally ordered — meaning that we might not be able to obtain bounds that are as tight as those possible using a weaker, partial order. It is possible to extend any partial order to a total order, but since this will result in a stronger set of constraints, we should expect to lose precision in the bounds. We will see in the next chapter — in Section 6.4 — how to modify this algorithm to work with a simple class of partial orders, so as to avoid this problem.

It is difficult to formally characterise the types of DTMC for which this algorithm works well — for example, the same model, with two different orderings of its state space, could result in bounds with very different precisions. We would expect a DTMC that is almost monotone, and almost lumpable to yield a better bound than one that is far from satisfying these properties, but there are no results that quantify this relationship formally. Certainly, manual applications of stochastic bounds can yield better results than the use of this algorithm, since they can be tailored to exploit particular structures in the model. The motivation in [69] was to present a general algorithm, that could be used in future to gain some insight into the types of model for which stochastic bounds works well, and to find heuristics for choosing a state space ordering.

In this chapter we have examined several state-based abstraction techniques for Markov chains. In particular, we looked at abstract Markov chains as an abstraction for transient properties of a Markov chain (namely, time-bounded reachability), and stochastic bounds as an abstraction for steady state properties. This is not to say that these techniques must be used exclusively for these purposes. Certainly, it should be possible to extend the technique of abstract Markov chains to bound long-run averages, using the techniques of [7], and stochastic bounds can be applied to any monotone property, meaning that these techniques could be adapted to time-bounded reachability.

We presented the techniques in this chapter in terms of Markov chains, but in practice we tend to use higher-level modelling formalisms that are more structured. In order to take advantage of this structure, we will look in the next chapter at extending these methods so that they can be applied *compositionally* to PEPA models. This will then lead to an implementation of a model checker and abstraction engine for PEPA, which we will demonstrate in Chapter 7.

Chapter 6

Stochastic Abstraction of PEPA Models

Stochastic abstraction techniques can be powerful when analysing large Markovian models, since they allow us to reduce a model to one that is small enough to analyse, whilst retaining information about the properties we are interested in. In the previous chapter, we introduced two particular techniques — abstract Markov chains and stochastic bounds — which allow us to reason about transient and steady state properties of Markov chains respectively. However, since these operate on the state space of the underlying Markov chain, they are limited in practice by the size of state space that we can represent.

When we describe a Markov chain using a compositional language such as PEPA, it leads to a concise description, even though the underlying state space might be enormous. In an ideal situation, we would like to analyse such models compositionally, so that we never have to deal with this state space explicitly. Yet unlike in the qualitative world, where there are many static analysis techniques for compositionally analysing programs [133], the availability of such techniques for stochastic systems is limited. Whilst there has been some work regarding compositional model checking [23], this relies on the model displaying a Boucherie product form structure [32], which is a very restrictive requirement in practice.

In this chapter, rather than looking at compositional *analysis*, we will instead develop techniques for compositional *abstraction*, in the context of PEPA models. In particular, we will show how to apply both abstract Markov chains and stochastic bounds compositionally, by making use of a Kronecker representation for PEPA [93]. By combining both techniques, we can model check all properties that can be specified in CSL/X, including both path and steady state formulae. The aim of this chapter is to establish and prove the safety of our abstractions — in the next chapter, we will describe a tool that we have developed, which implements these techniques.

We will begin this chapter with Section 6.1, where we show how the underlying Markov chain of a PEPA model can be described compositionally, in a Kronecker form. We will then describe our approach to abstracting and model checking PEPA models in Section 6.2, and show how to apply abstract Markov chains and stochastic bounds to PEPA in Sections 6.3 and 6.4 respectively. Proofs of the theorems in this chapter can be found in Appendix D.

6.1 Kronecker Representation of PEPA Models

Whilst in the previous chapter we looked at how to abstract Markov chains directly, the aim of this chapter is to present a *compositional* approach to abstraction. In our case, we will base this on the stochastic process algebra PEPA, which we introduced in Section 2.5.2. To apply the abstraction techniques of the previous chapter compositionally, we will first introduce a compositional representation for the generator matrix of the CTMC induced by PEPA's semantics. This approach is based on combining matrices using tensor, or *Kronecker* operators (see Appendix C) — this was first presented in [142] in the context of stochastic automata networks, and was first applied to stochastic process algebra in [35]. It was applied to PEPA in [93].

If we consider the system equation of a PEPA model, it has the following form:

$$C_1 \underset{L_1}{\bowtie} \cdots \underset{L_{N-1}}{\bowtie} C_N \tag{6.1}$$

Here, we ignore the hiding operator C/L without loss of generality, since it is always possible to rename action types to avoid name conflicts between components. Note that the cooperation combinator is *not* associative, however, which is an issue that we will return to in a moment.

The semantics of PEPA allows us to induce a CTMC from the system equation of a PEPA model. If we look at a *fragment* of the system equation, we can also induce a CTMC following the PEPA semantics — but only if the fragment cannot perform any passive activities. In order to describe the behaviour of a fragment that *can* perform passive activities, we will generalise the notion of a generator matrix. In particular, if we consider a sequential component C_i, having a state space $S_i = \mathrm{ds}(C_i)$, we can write a 'partial' generator matrix for the component as follows:

$$\boldsymbol{Q}_i = \sum_{a \in \mathcal{A}ct(C_i)} \boldsymbol{Q}_{i,a} = \sum_{a \in \mathcal{A}ct(C_i)} r_{i,a}\Big(\boldsymbol{P}_{i,a} - \boldsymbol{I}_{|S_i|}\Big) \tag{6.2}$$

Here, each $\boldsymbol{Q}_{i,a}$ is an $|S_i| \times |S_i|$ matrix that describes the behaviour of C_i due to activities of type a. Importantly, the elements of $\boldsymbol{Q}_{i,a}$ come from the set $\mathbb{R} \cup (\mathbb{R} \times \{\top\})$ — i.e. they correspond to either an active rate (in $\mathbb{R}$), or a passive rate (in $\mathbb{R} \times \{\top\}$). We define addition and multiplication over these elements as follows, for $r, s \in \mathbb{R}$:

$+$	s	$(s,\top)$
r	$r+s$	r
$(r,\top)$	s	$(r+s,\top)$

$\times$	s	$(s,\top)$
r	rs	$(rs,\top)$
$(r,\top)$	$(rs,\top)$	$(rs,\top)$

We further decompose each $\boldsymbol{Q}_{i,a}$ into a rate function $r_{i,a}$ and a probability transition matrix $\boldsymbol{P}_{i,a}$ — $r_{i,a} : S_i \to \mathbb{R}_{\geq 0} \cup \{\top\}$ gives the rate of action type a for each state in S_i, $\boldsymbol{P}_{i,a}$ gives the next-state transition probabilities conditional on performing an activity of type a, and $\boldsymbol{I}_{|S_i|}$ is the $|S_i| \times |S_i|$ identity matrix. If, for a state s, $r_{i,a}(s) = 0$, we write $\boldsymbol{P}_{i,a}(s,s) = 1$ and $\boldsymbol{P}_{i,a}(s,s') = 0$ for $s' \neq s$. Since the rate is zero, we could effectively have chosen any values for this row, but this choice is convenient since it encodes the fact that we remain in the same state[1]. Recall that the multiplication of a matrix by a rate function was defined in Equation 5.1.

As an example of such a 'partial' generator matrix, consider the following PEPA sequential component C_i:

$$\begin{aligned} C_i &\overset{def}{=} (a,r).C'_i \\ C'_i &\overset{def}{=} (b,0.5\top).C_i + (b,0.5\top).C'_i \end{aligned}$$

Here there are just two states, C_i and C'_i, and two action types, a and b, and the component of the generator matrix corresponding to C_i is as follows:

$$\begin{aligned} \boldsymbol{Q}_i &= \boldsymbol{Q}_{i,a} && + \ \boldsymbol{Q}_{i,b} \\ &= r_{i,a}(\boldsymbol{P}_{i,a} - \boldsymbol{I}_2) && + \ r_{i,b}(\boldsymbol{P}_{i,b} - \boldsymbol{I}_2) \\ &= \begin{bmatrix} r \\ 0 \end{bmatrix}\left(\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} - \boldsymbol{I}_2\right) && + \ \begin{bmatrix} 0 \\ \top \end{bmatrix}\left(\begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \end{bmatrix} - \boldsymbol{I}_2\right) \end{aligned}$$

To build a compositional representation of the generator matrix $\boldsymbol{Q}$ of an arbitrary PEPA model, whose system equation is structured as in Equation 6.1, we need to combine the individual generator matrices $\boldsymbol{Q}_{i,a}$ in an appropriate way. More precisely, the

[1] Note that this differs from [93, 94], where the elements of the row are all zero. In this case, instead of subtracting an identity matrix $\boldsymbol{I}$, they subtract a diagonal matrix $\boldsymbol{I}_{i,a}$ such that $\boldsymbol{I}_{i,a}(s,s) = 1$ if $r_{i,a}(s) > 0$, and is zero otherwise.

compositional representation of $\boldsymbol{Q}$ has to describe the same CTMC as induced by the semantics of the PEPA model. Because cooperation between two PEPA components uses the *minimum* of two rates, we need to be especially careful that this leads to the correct apparent rate for each state and action type.

To ensure this, and to provide a simple Kronecker form for the system equation, the use of *functional rates* for PEPA was proposed in [93]. This means that for each action type a, there is a single rate function r_a describing the apparent rate of a for each state in the system — which may depend on the state of more than one sequential component. For instance, consider the system equation of the example PEPA model from Figure 2.5:

$$Server \underset{\{request,\ response\}}{\bowtie} (Client \parallel Client) \tag{6.3}$$

Here, the ability of the system to perform a *request* activity depends on the local states of both clients *and* the state of the server, hence a functional rate for *request* must in general depend on the state of all three components.

So that we can avoid having functional rates that depend on multiple components in the model, we will introduce and use a variant of the Kronecker representation in [93, 94]. The difference is that we ensure that functional rates depend only on the state of a single component, at the expense of having more complicated combinators for combining the $\boldsymbol{Q}_{i,a}$ matrices. This leads to a representation that is a little less elegant mathematically, but which enables us to more easily establish and prove the results in the remainder of this chapter. The two main reasons are as follows:

1. We can store and abstract the apparent rate function of each component C_i by representing it as a vector of fixed size $|S_i|$, where S_i is the state space of C_i. This makes it more straightforward to construct our abstractions.

2. To prove the correctness of our compositional abstractions (Sections 6.3 and 6.4) we will just need to prove that safety is preserved by two Kronecker operators, which we will call $\circledast$ and $\odot$. These correspond respectively to cooperating and independent activities, and will be defined momentarily.

To describe the generator matrix term $\boldsymbol{Q}_{i,a}$ for activities of type a in a component C_i, we will use the shorthand $(r_{i,a}, \boldsymbol{P}_{i,a})$, which is defined as follows:

$$(r_{i,a}, \boldsymbol{P}_{i,a}) = r_{i,a}\left(\boldsymbol{P}_{i,a} - \boldsymbol{I}_{|S_i|}\right) = \boldsymbol{Q}_{i,a}$$

where S_i is the state space of C_i.

Recall that $r_{i,a}$ is an apparent rate function (depending only on the state of C_i) and $\boldsymbol{P}_{i,a}$ is a probabilistic transition matrix, as in Equation 6.2. If a component C_i cannot perform any activities of action type a, we define its generator matrix term to be $\boldsymbol{Q}_{i,a} = (r_\perp, \boldsymbol{I}_{|S_i|})$, where $r_\perp(s) = 0$ for all $s \in S_i$.

Using this notation, we can now introduce two Kronecker operators, $\circledast$ and $\odot$, which correspond to cooperating and independent activities. For cooperation over an action type a, we will use the operator $\circledast$, which is defined as follows[2]:

$$(r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a}) = (\min\{r_{1,a}, r_{2,a}\}, \boldsymbol{P}_{1,a} \otimes \boldsymbol{P}_{2,a}) \tag{6.4}$$

where $\min\{r_{1,a}, r_{2,a}\}(s_1, s_2) = \min\{r_{1,a}(s_1), r_{2,a}(s_2)\}$ for all $s_1 \in S_1$ and $s_2 \in S_2$. The operator $\otimes$ denotes the Kronecker product, and is defined in Appendix C.

If, on the other hand, activities of type a are performed independently, we will use the operator $\odot$, which we can define in terms of $\circledast$:

$$(r_{1,a}, \boldsymbol{P}_{1,a}) \odot (r_{2,a}, \boldsymbol{P}_{2,a}) = (r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_\top, \boldsymbol{I}_{|S_2|}) + (r_\top, \boldsymbol{I}_{|S_1|}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a}) \tag{6.5}$$

where $r_\top(s) = \top$ for all s. Intuitively, this is just a lifting of the Kronecker sum to our $(r, \boldsymbol{P})$ notation. Here, the '+' operator is standard matrix addition, but to continue to use our $(r, \boldsymbol{P})$ representation we will define it compositionally as follows:

Theorem 6.1.1. *Consider two generator matrices* $\boldsymbol{Q}_1 = (r_1, \boldsymbol{P}_1)$ *and* $\boldsymbol{Q}_2 = (r_2, \boldsymbol{P}_2)$*, corresponding to the same state space* S *—* $\boldsymbol{Q}_1$ *and* $\boldsymbol{Q}_2$ *are both* $|S| \times |S|$ *matrices. Then* $\boldsymbol{Q}_1 + \boldsymbol{Q}_2$ *can be written as follows:*

$$\boldsymbol{Q}_1 + \boldsymbol{Q}_2 = (r_1, \boldsymbol{P}_1) + (r_2, \boldsymbol{P}_2) = \left(r_1 + r_2, \frac{r_1}{r_1 + r_2}\boldsymbol{P}_1 + \frac{r_2}{r_1 + r_2}\boldsymbol{P}_2\right)$$

where $(r_1 + r_2)(s) = r_1(s) + r_2(s)$*, and* $\frac{r_i}{r_1+r_2}(s) = \frac{r_i(s)}{r_1(s)+r_2(s)}$*,* $i \in \{1, 2\}$*, for all* $s \in S$*.*

The coefficients of $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ describe the relative probability of taking a transition corresponding to $\boldsymbol{Q}_1$ or $\boldsymbol{Q}_2$. They are functions, because the relative apparent rate can differ between states — each row of the matrix might need to be multiplied by a different value.

For both of our Kronecker operators, $\circledast$ and $\odot$, the resulting generator matrix term is for the component $C_1 \underset{L}{\bowtie} C_2$, and has a state space of $S_1 \times S_2$. This Cartesian state space does not in general correspond to the derivative set $\mathrm{ds}(C_1 \underset{L}{\bowtie} C_2)$, since it may contain unreachable states. In practice, however, we never expand out the Kronecker

[2] A similar operator to $\circledast$ was used in [94], but for constant rates rather than rate functions.

form directly, in the sense of actually performing the tensor multiplications — instead, we derive only the *reachable* state space, using a bottom-up state space derivation algorithm, similar to that described in [177].

There are a number of memory-efficient algorithms for solving Markov chains at the Kronecker description level, in which the full generator matrix remains implicit [27, 64]. In our case, however, we view the Kronecker form just as an intermediate representation on which our abstractions are performed. Since we need to analyse not just Markov chains, but also *abstract* Markov chains, we do derive the state space of the model — but only *after* performing the abstraction. This is so that we can use the model checking algorithm in [18, 105], which works over an explicit representation of the state space.

We can now define our Kronecker representation for PEPA models, using the $\circledast$ and $\odot$ operators.

Definition 6.1.2. *Given a PEPA model $C = C_1 \underset{L_1}{\bowtie} \cdots \underset{L_{N-1}}{\bowtie} C_N$, its* Kronecker form $\boldsymbol{Q}(C)$ *is defined as follows:*

$$\boldsymbol{Q}(C_1 \underset{L_1}{\bowtie} \cdots \underset{L_{N-1}}{\bowtie} C_N) = \sum_{a \in \mathcal{A}ct(C)} \boldsymbol{Q}_a(C_1 \underset{L_1}{\bowtie} \cdots \underset{L_{N-1}}{\bowtie} C_N)$$

where $\mathcal{A}ct(C)$ is the set of all action types that occur in C (both synchronised and independent), and $\boldsymbol{Q}_a$ is defined inductively as follows:

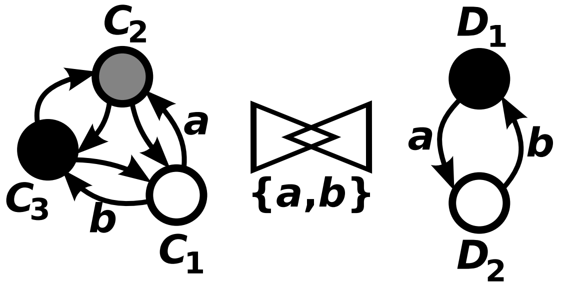
$$\begin{array}{lcll} \boldsymbol{Q}_a(C_i) & = & (r_{i,a}, \boldsymbol{P}_{i,a}) & \text{if } C_i \textit{ is a sequential component} \\ \boldsymbol{Q}_a(C_i \underset{L}{\bowtie} C_j) & = & \left\{ \begin{array}{l} \boldsymbol{Q}_a(C_i) \circledast \boldsymbol{Q}_a(C_j) \\ \boldsymbol{Q}_a(C_i) \odot \boldsymbol{Q}_a(C_j) \end{array} \right. & \begin{array}{l} \text{if } a \in L \\ \text{if } a \notin L \end{array} \end{array}$$

The following theorem establishes the correctness of our Kronecker representation, in that it defines an equivalent CTMC to that induced by the PEPA semantics:

Theorem 6.1.3. *For all well-formed*[3] *PEPA models C, the CTMC induced by the semantics of PEPA and the CTMC described by the generator matrix $\boldsymbol{Q}(C)$, projected onto the derivative set* ds(C) *(the reachable state space of C), are isomorphic.*

As an example of how the Kronecker form is applied, let us take the PEPA model in Figure 6.1. Here, there are two sequential components (C_1 and D_1) and three action

[3] A well-formed PEPA model is one in which cooperation occurs only at the level of the system equation. If a model has a single system equation, the PEPA syntax given in Section 2.5.2 implicitly guarantees that it is well-formed.



$$
\begin{array}{lcl@{\qquad}lcl}
C_1 & \overset{def}{=} & (a,r_a).C_2 + (b,r_b).C_3 & D_1 & \overset{def}{=} & (a,r_D).D_2 \\
C_2 & \overset{def}{=} & (\tau,r_2).C_1 + (\tau,r_2).C_3 & D_2 & \overset{def}{=} & (b,r_D).D_1 \\
C_3 & \overset{def}{=} & (\tau,r_3).C_2 + (\tau,r_3).C_1 & & &
\end{array}
$$

$$C_1 \underset{\{a,b\}}{\bowtie} D_1$$

Figure 6.1: An example PEPA model and its graphical representation

types — we cooperate over a and b, but τ is performed independently. Applying our Kronecker form, we arrive at the following structure for $\boldsymbol{Q}(C_1 \underset{\{a,b\}}{\bowtie} D_1)$:

$$
\begin{array}{rclcl}
\boldsymbol{Q}(C_1 \underset{\{a,b\}}{\bowtie} D_1) & = & \boldsymbol{Q}_\tau(C_1) & \odot & \boldsymbol{Q}_\tau(D_1) \\
& + & \boldsymbol{Q}_a(C_1) & \circledast & \boldsymbol{Q}_a(D_1) \\
& + & \boldsymbol{Q}_b(C_1) & \circledast & \boldsymbol{Q}_b(D_1)
\end{array}
$$

If we had an additional copy of component D, such that the system equation was $C_1 \underset{\{a,b\}}{\bowtie} (D_1 \parallel D_1)$, then $\boldsymbol{Q}(C_1 \underset{\{a,b\}}{\bowtie} (D_1 \parallel D_1))$ would be written as:

$$
\begin{array}{rclcl}
\boldsymbol{Q}(C_1 \underset{\{a,b\}}{\bowtie} (D_1 \parallel D_1)) & = & \boldsymbol{Q}_\tau(C_1) & \odot & (\boldsymbol{Q}_\tau(D_1) \odot \boldsymbol{Q}_\tau(D_1)) \\
& + & \boldsymbol{Q}_a(C_1) & \circledast & (\boldsymbol{Q}_a(D_1) \odot \boldsymbol{Q}_a(D_1)) \\
& + & \boldsymbol{Q}_b(C_1) & \circledast & (\boldsymbol{Q}_b(D_1) \odot \boldsymbol{Q}_b(D_1))
\end{array}
$$

Returning to our model with just two components, let us consider the internal action type τ of component C. We can write the corresponding generator matrix term, $\boldsymbol{Q}_\tau(C_1) = (r_{C,\tau}, \boldsymbol{P}_{C,\tau})$ as follows (where index i represents the state C_i, for $1 \leq i \leq 3$):

$$
\boldsymbol{Q}_\tau(C_1) = \left(\begin{bmatrix} 0 \\ 2r_2 \\ 2r_3 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 2r_2 \\ 2r_3 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right)
$$

Although it has been written as a vector in the above, it is important to remember that the rate function is a *function*, and is interpreted as multiplying each row of the probability transition matrix by the corresponding rate. The generator matrix for the entire model can be written in its Kronecker form as follows, where we expand out the $\circledast$ and $\odot$ operators to show the tensor products $\otimes$:

$$
\begin{aligned}
\boldsymbol{Q} = \min&\left\{\begin{bmatrix} 0 \\ 2r_2 \\ 2r_3 \end{bmatrix}, \begin{bmatrix} \top \\ \top \end{bmatrix}\right\}\left(\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\
+ \min&\left\{\begin{bmatrix} \top \\ \top \\ \top \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right\}\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\
+ \min&\left\{\begin{bmatrix} r_a \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_D \\ 0 \end{bmatrix}\right\}\left(\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\
+ \min&\left\{\begin{bmatrix} r_b \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ r_D \end{bmatrix}\right\}\left(\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)
\end{aligned}
\tag{6.6}
$$

Note that the second term in the above evaluates to zero, because the D component does not perform any internal τ activities.

6.2 Compositional Abstraction of PEPA Models

In Section 5.3 of the previous chapter we introduced abstract Markov chains and stochastic comparison in the context of CTMCs, and in the previous section we have seen how the CTMC generated by a PEPA model can be described compositionally. We could naïvely apply these bounding techniques to PEPA models by first deriving the underlying Markov chain, but this would not take advantage of the compositional structure of the model. Moreover, in order to apply the bounding techniques, we would first need to generate and store the concrete Markov chain — if the model is sufficiently complex, this might not be possible.

The purpose of the remainder of this chapter is to show how we can directly apply the bounding techniques we have seen to PEPA models, in a *compositional* manner. The basic idea is illustrated in Figure 6.2 — we abstract each sequential component of a PEPA model separately, in such a way that the CTMC derived from the abstract model is a *safe abstraction* of the CTMC derived from the concrete model. What we mean by 'safe' depends on the abstraction technique — we will discuss this in detail for abstract Markov chains and stochastic bounds in Sections 6.3 and 6.4 respectively.

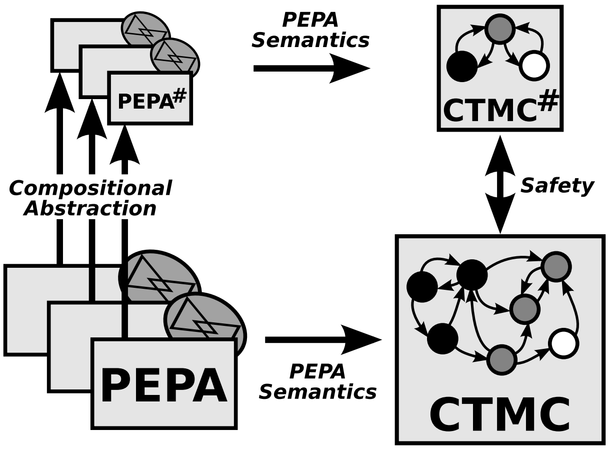


Figure 6.2: Compositional abstraction of PEPA models

To apply these techniques *compositionally*, we need to have defined an abstraction $(S_i^\sharp, \alpha_i)$ for each sequential component C_i of a PEPA model, specifying how we want to aggregate its states. This induces an abstraction $(S^\sharp, \alpha)$ over the state space of the system. Of course, we need to be able to define such an abstraction for this to be practical, and we show how to do so using our tool in the next chapter — finding a good abstraction automatically is difficult, and so we instead provide an interface for graphically selecting states to aggregate. As an example, consider component C of the PEPA model in Figure 6.1. We can abstract its concrete state space $S_C = \{C_1, C_2, C_3\}$ to $S_C^\sharp = \{C_1, C_{\{2,3\}}\}$, using the following abstraction function:

$$\alpha(C_1) = C_1 \qquad \alpha(C_2) = C_{\{2,3\}} \qquad \alpha(C_3) = C_{\{2,3\}}$$

The reason for using both abstract Markov chains *and* stochastic bounds, is that we want to be able to model check both path properties and steady state properties of CSL/X. We discussed the bounding of CSL path properties in Section 5.6.1, in relation to model checking the until operator of three-valued CSL for abstract CTMCs. We also described in Section 5.6.2 how stochastic bounds can be used to bound the steady state distribution of a CTMC. If we combine these two approaches we can model check all properties in CSL/X.

Consider a CSL/X steady state property, which has the following form, in the case of a probability test[4], where Φ is itself a CSL/X state property:

$$\mathcal{S}_{=?}(\Phi)$$

[4] We could similarly have asked whether the probability is above or below a certain value, for example $\mathcal{S}_{<p}(\Phi)$, but we would still need to compute the interval on the probability first.

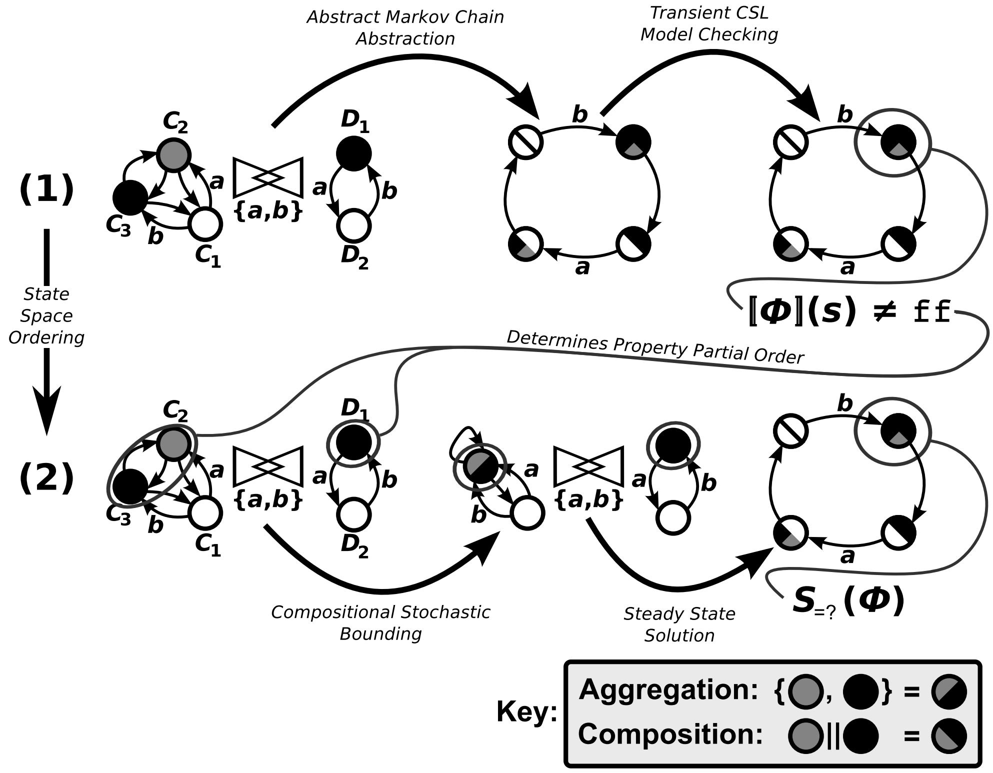


Figure 6.3: Model checking of CSL steady state properties

This asks the question: "what is the probability, in the steady state of our model, that we will be in a state that satisfies Φ?" In general, the answer is a probability interval, since an abstraction usually introduces some uncertainty.

There are two stages to model checking such a property, which we illustrate in Figure 6.3, using the example from PEPA model from Figure 6.1. These are as follows:

1. We find upper and lower bounds for the set of states that satisfy Φ. We can assume without loss of generality that Φ does not itself contain a steady state operator — if it does, we can apply an inductive argument to model check Φ. However, if the Markov chain is ergodic, then it does not make sense to nest steady state formulae [23].

   Since Φ is then a CSL/X property not including the steady state operator, we use the technique of abstract Markov chains to compute the following sets of states (we will show how to do this in Section 6.3):

$$\begin{aligned} S_\Phi^L &= \{s \mid [\![\Phi]\!](s) = \texttt{tt}\} \\ S_\Phi^U &= \{s \mid [\![\Phi]\!](s) \neq \texttt{ff}\} \end{aligned} \tag{6.7}$$

where $[\![\Phi]\!](s)$ denotes the (three-valued) truth of property Φ in state s (see Figure 5.4). The lower-bounding set S^L_Φ contains those states that *definitely* satisfy Φ, and the upper-bounding set S^U_Φ contains all states that *might* satisfy Φ.

2. We find upper and lower stochastic bounds for the PEPA model, from which we can obtain probability bounds for the steady state property. For the lower bound, we want an under-approximation of the probability of being in S^L_Φ, and for the upper bound we want an over-approximation of the probability of being in S^U_Φ. To do this, we compositionally construct a lower and upper bound of the PEPA model, using a partial order derived from the sets S^L_Φ and S^U_Φ respectively. We will show how to construct this partial order and the stochastic bounds in Section 6.4. From this, we can solve the bounding PEPA models to obtain bounding steady state distributions, which give us upper and lower bounds for the quantitative CSL property $\mathcal{S}_{=?}(\Phi)$.

Figure 6.3 shows these two stages when computing an upper bound for the probability of satisfying a state formula Φ, given an abstraction that combines the states C_1 and C_2 of the original model. In the first stage, we compute S^U_Φ — the set of states that possibly satisfy Φ — using a compositional abstract Markov chain abstraction, and three-valued CSL model checking. We can then calculate the steady state probability $\mathcal{S}_{=?}(\Phi)$ in the second stage, on a stochastic bound of the model.

In order to *compositionally* compute a stochastic bound of the PEPA model, however, we first need to express the set S^U_Φ compositionally — namely, we need to have a separate set of states for each component. For a model of N components, the upper-bounding set of states for the ith component is given by S^U_i:

$$S^U_i = \left\{ s_i \;\middle|\; (s_1, \ldots, s_i, \ldots, s_N) \in S^U_\Phi \right\} \tag{6.8}$$

In general this leads to an over-approximation of the set S^U_Φ, but we gain compositionality. In the example, $S^U_\Phi = \{(C_2, D_1), (C_3, D_1)\}$, which leads to $S^U_C = \{C_2, C_3\}$ and $S^U_D = \{D_1\}$. In the second stage we construct a compositional stochastic bounding PEPA model, using an ordering that allows us to compare the sets of states S^U_C and S^U_D. Finally, we construct and solve the CTMC induced by the PEPA semantics.

The remaining sections in this chapter are concerned with applying abstract Markov chains and stochastic bounds to PEPA models compositionally, so that we can use the above approach. In particular, the above discussion assumes that we already know how to apply both techniques, and shows how we can combine them. Note once again that our approach is to use compositional abstraction, rather than performing the model checking compositionally[5], in the sense of [23]. We have implemented a tool that allows the specification and model checking of CSL properties on PEPA models, which we will see in more detail, along with a number of examples, in Chapter 7.

6.3 Model Checking of Transient Properties

To model check transient CSL properties, excluding the timed next operator, we will show in this section how to compositionally construct an abstract CTMC from the Kronecker representation of a PEPA model. We will begin by defining an *abstract CTMC component*, in which we bound the probability transition matrix and the rate function separately.

Definition 6.3.1. *An* abstract CTMC component *is a tuple* $(S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^L, \boldsymbol{P}^U, r^L, r^U, L^\sharp)$*, where* $S^\sharp$*,* $\boldsymbol{\pi}^{(0)\sharp}$*,* $\boldsymbol{P}^L$*,* $\boldsymbol{P}^U$*, and* $L^\sharp$ *are defined the same as for an abstract CTMC, and* $r^L, r^U : S^\sharp \rightarrow \mathbb{R}_{\geq 0} \cup \{\top\}$ *are functions such that* $r^L(s) \leq r^U(s)$ *for all* $s \in S^\sharp$*.*

An abstract CTMC component induces an abstract CTMC as follows:

Definition 6.3.2. *Let* $\mathcal{M}^{\sharp\sharp} = (S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}_a^L, \boldsymbol{P}_a^U, r_a^L, r_a^U, L^\sharp)$ *be an abstract CTMC component. Given a unformisation constant* $\lambda \geq \max_{s \in S^\sharp} r_a^U(s)$*, we can construct an abstract CTMC as follows:*

$$ACTMC_\lambda(\mathcal{M}^{\sharp\sharp}) = (S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^L, \boldsymbol{P}^U, \lambda, L^\sharp)$$

where $\boldsymbol{P}^L$ *and* $\boldsymbol{P}^U$ *are defined as follows:*

$$\begin{aligned} \boldsymbol{P}^L(s,s') &= \begin{cases} \frac{r_a^L(s)}{\lambda}\boldsymbol{P}_a^L(s,s') & \text{if } s \neq s' \\ \frac{r_a^L(s)}{\lambda}\boldsymbol{P}_a^L(s,s) + \left(1 - \frac{r_a^U(s)}{\lambda}\right) & \text{otherwise} \end{cases} \\ \boldsymbol{P}^U(s,s') &= \begin{cases} \frac{r_a^U(s)}{\lambda}\boldsymbol{P}_a^U(s,s') & \text{if } s \neq s' \\ \frac{r_a^U(s)}{\lambda}\boldsymbol{P}_a^U(s,s) + \left(1 - \frac{r_a^L(s)}{\lambda}\right) & \text{otherwise} \end{cases} \end{aligned}$$

The intuition here is that we add the diagonal elements to account for the term $1 - \frac{r_a}{\lambda}\boldsymbol{I}$ that appears in the uniformised probabilistic transition matrix:

$$\boldsymbol{P} = \frac{1}{\lambda}\boldsymbol{Q} + \boldsymbol{I} = \frac{1}{\lambda}r_a(\boldsymbol{P}_a - \boldsymbol{I}) + \boldsymbol{I}$$

Since we only have upper and lower bounds for the rates r_a, we need to choose the most conservative values to ensure that the bound is correct. This comes at a loss of

[5] The exception being for atomic properties that are already specified compositionally.

precision, but this is necessary if we are to combine the abstract CTMC components and still end up with a safe abstract CTMC — with respect to the abstract CTMC obtained from the Markov chain of the PEPA model. In this context, an abstract CTMC $\mathcal{M}_2^\sharp$ is a safe approximation of $\mathcal{M}_1^\sharp$ if $\mathcal{M}_1^\sharp \leq \mathcal{M}_2^\sharp$, as per Definition 5.6.2.

Given a sequential PEPA component C_i with state space S_i, we can define a CTMC $\mathcal{M}_{i,a} = (S_i, \boldsymbol{\pi}_i^{(0)}, \boldsymbol{P}_{i,a}, r_{i,a}, L_i)$ describing the behaviour of the component with respect to action type a. This will not necessarily be ergodic, since some states of C_i might not perform an action of type a. The component $\boldsymbol{Q}_{i,a}$ corresponding to $\mathcal{M}_{i,a}$ in the Kronecker representation of the PEPA model is defined as $\boldsymbol{Q}_{i,a} = r_{i,a}(\boldsymbol{P}_{i,a} - \boldsymbol{I})$. From the CTMC $\mathcal{M}_{i,a}$, given an abstraction $(S_i^\sharp, \alpha_i)$, we can derive an abstract CTMC component as follows:

Definition 6.3.3. *The abstract CTMC component induced by an abstraction $(S^\sharp, \alpha)$ on a CTMC $\mathcal{M} = (S, \boldsymbol{P}, r, L)$ is defined as:*

$$AbsComp_{(S^\sharp,\alpha)}(\mathcal{M}) = (S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^L, \boldsymbol{P}^U, r^L, r^U, L^\sharp)$$

where:

$$\begin{aligned}
\boldsymbol{\pi}^{(0)\sharp}(s^\sharp) &= \sum_{s\in\gamma(s^\sharp)} \boldsymbol{\pi}^{(0)}(s) \\
\boldsymbol{P}^L(s_1^\sharp, s_2^\sharp) &= \min_{s_1\in\gamma(s_1^\sharp)} \sum_{s_2\in\gamma(s_2^\sharp)} \boldsymbol{P}(s_1, s_2) \\
\boldsymbol{P}^U(s_1^\sharp, s_2^\sharp) &= \max_{s_1\in\gamma(s_1^\sharp)} \sum_{s_2\in\gamma(s_2^\sharp)} \boldsymbol{P}(s_1, s_2) \\
r^L(s^\sharp) &= \min_{s\in\gamma(s^\sharp)} r(s) \\
r^U(s^\sharp) &= \max_{s\in\gamma(s^\sharp)} r(s) \\
L^\sharp(s^\sharp, a) &= \begin{cases} \mathtt{tt} & \text{if } \forall s \in \gamma(s^\sharp).\ L(s,a) = \mathtt{tt} \\ \mathtt{ff} & \text{if } \forall s \in \gamma(s^\sharp).\ L(s,a) = \mathtt{ff} \\ ? & \text{otherwise} \end{cases}
\end{aligned}$$

Theorem 6.3.4. *Consider a CTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$. For any uniformisation constant $\lambda \geq \max_{s\in S} r(s)$, and any abstraction $(S^\sharp, \alpha)$ on $\mathcal{M}$, the following holds:*

$$Abs_{(S^\sharp,\alpha)}(Unif_\lambda(\mathcal{M})) \leq ACTMC_\lambda\left(AbsComp_{(S^\sharp,\alpha)}(\mathcal{M})\right)$$

This theorem states that abstract CTMC components *safely approximate* abstract CTMCs. In other words, an abstract CTMC component gives an over-approximation of the probability transition intervals, compared to directly generating an abstract CTMC.

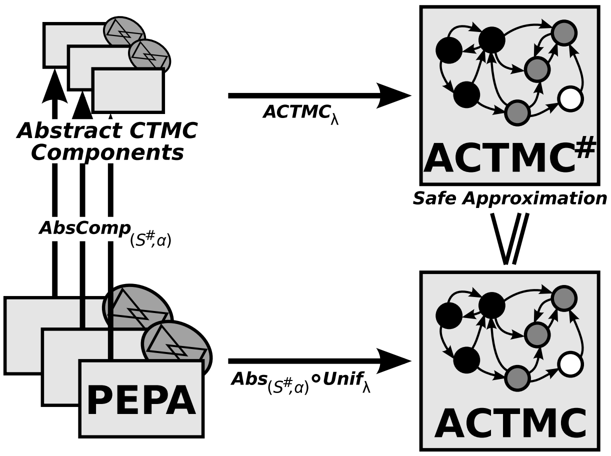


Figure 6.4: Safety property of Abstract CTMC Components (Theorem 6.3.5)

Consider two abstract CTMC components, $\mathcal{M}^{\sharp\sharp}_{1,a} = (S^\sharp_1, \boldsymbol{\pi}^{(0)\sharp}_1, \boldsymbol{P}^L_{1,a}, \boldsymbol{P}^U_{1,a}, r^L_{1,a}, r^U_{1,a}, L^\sharp_1)$ and $\mathcal{M}^{\sharp\sharp}_{2,a} = (S^\sharp_2, \boldsymbol{\pi}^{(0)\sharp}_2, \boldsymbol{P}^L_{2,a}, \boldsymbol{P}^U_{2,a}, r^L_{2,a}, r^U_{2,a}, L^\sharp_2)$. We can construct a new abstract CTMC component, corresponding to the two components cooperating over action type a as follows:

$$\mathcal{M}^{\sharp\sharp}_{1,a} \circledast \mathcal{M}^{\sharp\sharp}_{2,a} = \Big(S^\sharp_1 \times S^\sharp_2, \boldsymbol{\pi}^{(0)\sharp}_1 \otimes \boldsymbol{\pi}^{(0)\sharp}_2, \boldsymbol{P}^L_{1,a} \otimes \boldsymbol{P}^L_{2,a}, \boldsymbol{P}^U_{1,a} \otimes \boldsymbol{P}^U_{2,a}, \min\{r^L_{1,a}, r^L_{2,a}\}, \min\{r^U_{1,a}, r^U_{2,a}\}, L^\sharp_1 \times L^\sharp_2\Big)$$

where the new labelling function is $L^\sharp_1 \times L^\sharp_2((s_1,s_2),a) = L^\sharp_1(s_1,a) \wedge L^\sharp_2(s_2,a)$. The minimum operators in the above come from the semantics of cooperation in PEPA, so they apply to both the upper and lower bounds for the rates. If the two components do not cooperate over the action type a (i.e. they perform activities of type a independently), then the new abstract CTMC component will instead be:

$$\mathcal{M}^{\sharp\sharp}_{1,a} \odot \mathcal{M}^{\sharp\sharp}_{2,a} = \Big(S^\sharp_1 \times S^\sharp_2, \boldsymbol{\pi}^{(0)\sharp}_1 \otimes \boldsymbol{\pi}^{(0)\sharp}_2, \boldsymbol{P}^L_{1,a} \oplus \boldsymbol{P}^L_{2,a}, \boldsymbol{P}^U_{1,a} \oplus \boldsymbol{P}^U_{2,a}, r^L_{1,a} + r^L_{2,a}, r^U_{1,a} + r^U_{2,a}, L^\sharp_1 \times L^\sharp_2\Big)$$

where $(r^B_{1,a} + r^B_{2,a})(s_1,s_2) = r^B_{1,a}(s_1) + r^B_{2,a}(s_2)$ for $B \in \{L, U\}$.

We can now present the main theorem of this section — that the abstract CTMC we obtain by composing the abstract CTMC components of a PEPA model is a safe approximation of the abstract CTMC obtained by abstracting the CTMC induced by the PEPA semantics. This is illustrated in Figure 6.4.

Theorem 6.3.5. *Consider two PEPA components C_1 and C_2, with abstractions $(S^\sharp_1, \alpha_1)$ and $(S^\sharp_2, \alpha_2)$ respectively. Let $\mathcal{M}^{\sharp\sharp}_{i,a} = AbsComp_{(S^\sharp_i,\alpha_i)}(\boldsymbol{Q}_a(C_i))$ for $i \in \{1,2\}$. Then for* *all λ such that $Unif_\lambda(C_1 \underset{L}{\bowtie} C_2)$ is defined, the following holds:*

$$Abs_{(S^\sharp,\alpha)}\Big(Unif_\lambda\Big(\boldsymbol{Q}\Big(C_1 \underset{L}{\bowtie} C_2\Big)\Big)\Big) \leq ACTMC_\lambda\left(\sum_{a\in L}\mathcal{M}^{\sharp\sharp}_{1,a}\circledast\mathcal{M}^{\sharp\sharp}_{2,a}+\sum_{a\in\overline{L}}\mathcal{M}^{\sharp\sharp}_{1,a}\odot\mathcal{M}^{\sharp\sharp}_{2,a}\right)$$

where $S^\sharp = S^\sharp_1 \times S^\sharp_2$, $\alpha(s_1,s_2) = (\alpha_1(s_1),\alpha_2(s_2))$, and $\overline{L} = (\mathcal{A}ct(C_1)\cup\mathcal{A}ct(C_2))\setminus L$.

Since this method produces an over-approximation to the non-compositionally derived abstract CTMC, we can directly apply the model checking algorithm described in [18, 105] to this abstract Markov chain — this allows us to check transient three-valued CSL properties. In particular, given a CSL state property Φ, we can determine the set of states that definitely satisfy Φ (the model checker returns `tt`), and those that definitely do not satisfy Φ (the model checker returns `ff`).

Note that we do not consider the PEPA hiding operator here, since it is always possible to eliminate hiding by alpha-renaming of action types. This is because, in a well-formed PEPA model, hiding can only occur at the system equation level — hence an action type cannot be dynamically hidden during the model's execution.

As an example, let us return to the PEPA model from Figure 6.1, and consider constructing an abstract CTMC for the case when we aggregate states C_2 and C_3. The Kronecker form of the generator matrix $\boldsymbol{Q}$ of the model is as follows (the same as Equation 6.6, but without the term that evaluates to zero). To make the example concrete, we set the rates such that $r_a = r_2 = r_D = 1$ and $r_b = r_3 = 2$:

$$\begin{aligned}
\boldsymbol{Q} = \min\left\{\begin{bmatrix}0\\2\\4\end{bmatrix},\begin{bmatrix}\top\\\top\end{bmatrix}\right\}&\left(\begin{bmatrix}1&0&0\\\frac{1}{2}&0&\frac{1}{2}\\\frac{1}{2}&\frac{1}{2}&0\end{bmatrix}\otimes\begin{bmatrix}1&0\\0&1\end{bmatrix}-\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\otimes\begin{bmatrix}1&0\\0&1\end{bmatrix}\right)\\
+\ \min\left\{\begin{bmatrix}1\\0\\0\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right\}&\left(\begin{bmatrix}0&1&0\\0&1&0\\0&0&1\end{bmatrix}\otimes\begin{bmatrix}0&1\\0&1\end{bmatrix}-\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\otimes\begin{bmatrix}1&0\\0&1\end{bmatrix}\right)\\
+\ \min\left\{\begin{bmatrix}2\\0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right\}&\left(\begin{bmatrix}0&0&1\\0&1&0\\0&0&1\end{bmatrix}\otimes\begin{bmatrix}1&0\\1&0\end{bmatrix}-\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\otimes\begin{bmatrix}1&0\\0&1\end{bmatrix}\right)
\end{aligned}$$

The compositional abstract Markov chain now has the following form, where we save space by writing intervals for the elements of the matrices, rather than intervals on the matrices themselves:

$$
\begin{aligned}
\boldsymbol{Q}^{\sharp} = \min\left\{\begin{bmatrix} 0 \\ [2,4] \end{bmatrix}, \begin{bmatrix} \top \\ \top \end{bmatrix}\right\} & \left(\begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\
+ \min\left\{\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right\} & \left(\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\
+ \min\left\{\begin{bmatrix} 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right\} & \left(\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)
\end{aligned}
$$

We can multiply this out to arrive at the following abstract CTMC (where the uniformisation constant $\lambda = 4$). For clarity, we have labelled the state that each row of the matrix corresponds to:

$$
\boldsymbol{Q}^{\sharp} = \begin{array}{c} (C_1 D_1) \\ (C_1 D_2) \\ (C_{\{2,3\}} D_1) \\ (C_{\{2,3\}} D_2) \end{array} 4\left(\begin{bmatrix} \frac{3}{4} & 0 & 0 & \frac{1}{4} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ [\frac{1}{4}, \frac{1}{2}] & 0 & [\frac{1}{2}, \frac{3}{4}] & 0 \\ 0 & [\frac{1}{4}, \frac{1}{2}] & 0 & [\frac{1}{2}, \frac{3}{4}] \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\right)
$$

We are now in a position to model check transient CSL/X properties of this abstract CTMC, and compare them to the original PEPA model. As an example, consider the property $\mathcal{P}_{=?}(C_1 \ \mathcal{U}^{[0,1]} \ C_{\{2,3\}})$, which asks the question "what is the probability that within the first time unit, we will remain in state C_1 before moving to state C_2 or C_3?" Since there are only two states in the abstracted C component, this is equivalent to asking whether we will leave state C_1 within the first time unit. Model checking the original model gives an answer of 0.6321, and in this case the abstract CTMC gives a precise answer of $[0.6321, 0.6321]$.

In this case, the abstraction is a success because it gives a very tight bound on the property. Of course, in general we cannot expect to always obtain tight bounds, and the choice of abstraction has a large impact on the precision. The purpose of this small example was to demonstrate how our abstraction is applied — we will look at some larger examples in the next chapter, which better illustrate how this technique can be useful in practice.

6.4 Model Checking of Steady State Properties

We will now turn to model checking steady state properties of PEPA models, for which we apply the technique of stochastic bounds that was introduced in Section 5.6.2. The work that we present here is general in the sense that it applies to *all* well-formed PEPA models. This is in contrast to previous work, which considered the application of stochastic bounds to particular classes of PEPA model, such as passage time properties of workflow-structured models [68]. There is an advantage to looking at specific classes of model, in that it may be possible to obtain more accurate bounds in light of the additional information that is available. However, it is clear in our context that generality is important, since we want to avoid any restrictions on the structure of models that we extract from program code — in particular, allowing the model-level transformations that we introduced in Section 4.6. We will therefore present a general approach to bounding steady state properties of PEPA models, that can be applied to the models produced by the techniques of Chapter 4.

If we recall the approach that we described for computing CSL steady state properties, we need to find an upper bound for the steady state probability of being in a set of states S^U and a lower bound for the steady state probability of being in a set of states S^L, as defined in Equation 6.7. Let us consider the upper bound — the set S^U is not specified compositionally in terms of the states of each sequential component, hence we need to first convert it into a compositionally specified set. This is so that we can define a partial order on the state space, and so construct a stochastic bound of each component separately. There are two approaches we can take to doing this:

1. Find the smallest over-approximation of the set S^U that can be specified compositionally (see Equation 6.8).

2. Separate S^U into disjoint sets that can each be specified compositionally.

For example, consider the PEPA model from Figure 6.1. If the property is $S^U = \{(C_1,D_1),(C_2,D_2)\}$ then the first method would give $\{C_1,C_2\}\times\{D_1,D_2\} = \{(C_1,D_1),(C_2,D_1),(C_1,D_2),(C_2,D_2)\}$, which gives a coarser approximation to the set of states. Conversely, the second method would give two sets, $\{C_1\}\times\{D_1\} = \{(C_1,D_1)\}$ and $\{C_2\}\times\{D_2\} = \{(C_2,D_2)\}$, which together specify the same set as S^U, but require us to produce two different stochastic bounds of the model.

Before bounding a PEPA model, we need to decide upon two things — an *ordering*, and a *partitioning* of its state space. Since the idea is to produce the bound compositionally, these must also be defined compositionally. Finding a good ordering and partitioning is difficult — we rely on the user to specify the partitioning (see Chapter 7 for details on how this is done), although we choose the ordering automatically.

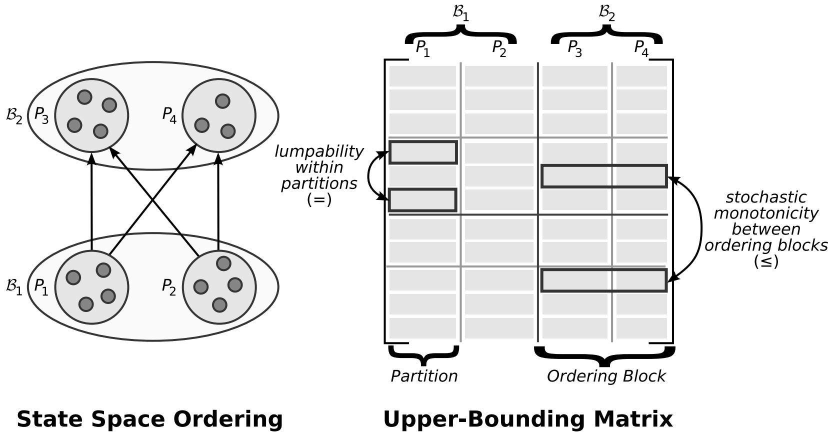


Figure 6.5: State space ordering and lumpability constraints

In Section 6.2 we stated that for a PEPA model with N components, $C_1,\ldots,C_N$, we define an abstraction $(S_i^\sharp,\alpha_i)$ over the state space S_i of each component C_i. Together, these induce an abstraction over the Cartesian state space $S_1\times\cdots\times S_N$ of the system. This defines a unique partitioning of the state space according to which concrete states map to the same abstract state.

In addition to partitioning the state space, we need to provide an ordering. The ordering we choose will in general depend on the property we are interested in. For example, if we are interested in the steady state probability of being in a particular set of states, it makes sense to place these at the 'top' of the ordering. This is so that we can directly compare the probabilities of being in this set. Furthermore, choosing a partial order can be advantageous, since it allows more flexibility when constructing the bound. The only constraint we have is that we only allow entire partitions to be compared with other partitions, so that the abstraction can be applied.

The definitions and theorems in this section are applicable to any partial order, but in order to algorithmically construct the bound, we will restrict ourselves to the following class:

Definition 6.4.1. *A* simple partial order *over a state space S is given by a set of M disjoint sets, $\mathcal{B}_1,\ldots,\mathcal{B}_M\subseteq S$, such that $\cup_i\mathcal{B}_i = S$, where:*

$$s \prec s' \ \text{ iff } \ \exists i,j.\ s\in\mathcal{B}_i \wedge s'\in\mathcal{B}_j \wedge i<j$$

Each $\mathcal{B}_i$ may contain multiple partitions, but not vice versa. This is illustrated in Figure 6.5, which shows an example state space ordering and partitioning, and the resulting constraints on the upper-bounding transition matrix.

Most of the time, however, we are only interested in the probability of being in *one* particular set of states — for example the set S_Φ of states that satisfy the CSL state formula Φ. Under these circumstances, we have a special case of the simple partial order:

Definition 6.4.2. *The* property partial order *of a set of states $S_\Phi \subseteq S$ over the state space S of a PEPA component is defined as follows:*

$$s \prec s' \text{ iff } s \in S_\Phi^C \wedge s' \in S_\Phi$$

where S_Φ^C is the complement of S_Φ.

This is an instance of the simple partial order, when $M = 2$, $\mathcal{B}_1 = S_\Phi^C$, and $\mathcal{B}_2 = S_\Phi$. For the remainder of this section, we will present our results in relation to the simple partial order, which is the more general of the two. For implementation as part of a CSL model checker, however, the property partial order is sufficient.

It is important to point out that choice of ordering can significantly affect the precision of the bounds. The property partial order that we define here is a *heuristic*, in that we can expect it to give the best result in many cases, due to imposing the fewest constraints on the stochastic bound. For particular models, however, there may be other orderings that provide greater precision, due to exploiting some structure in the model.

6.4.1 Stochastic Bounding of PEPA Models

Given an ordering and partitioning of a state space, we need to find a monotone CTMC that is both lumpable and an upper bound of the original CTMC. To do this compositionally, we must work at the level of the matrices $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$, bounding both the rate function r_a and the transition matrix $\boldsymbol{P}_a$ separately. This is so that when we construct the generator matrix of the entire model, by expanding out the Kronecker form in Definition 6.1.2, it remains upper-bounding, monotone and lumpable.

Unfortunately, it is not the case that the monotonicity of r_a and $\boldsymbol{P}_a$ implies that of $\boldsymbol{Q}_a$. In order for $\boldsymbol{Q}_a$ to be monotone, its embedded DTMC after we uniformise it must be monotone, as per Definition 5.6.10. The effect of uniformisation is to add self loops (i.e. to add probability mass to the diagonal elements) so that each state has the same exit rate λ. Since r_a is *increasing*, however, this means that we add a *decreasing* amount to the diagonal element of each row (the $-r_a\boldsymbol{I}$ term).

This is best illustrated by example. In the following, both the rate function r_a and the probability transition matrix $\boldsymbol{P}_a$ are monotone (assuming a totally ordered state space), but the embedded DTMC after uniformisation is not:

$$r_a(\boldsymbol{P}_a - \boldsymbol{I}) = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \left(\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} - \boldsymbol{I} \right) = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \left(\begin{bmatrix} \frac{3}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} - \boldsymbol{I} \right) \tag{6.9}$$

To avoid this problem, we need to strengthen the definitions of stochastic ordering and monotonicity, by adding an extra constraint. We call these the *rate-wise stochastic ordering* and *rate-wise monotonicity* respectively. Their definitions are:

Definition 6.4.3. *Given two generator matrix components $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ and $\boldsymbol{Q}'_a = r'_a(\boldsymbol{P}'_a - \boldsymbol{I})$, we say that $\boldsymbol{Q}_a \leq_{\mathrm{rst}} \boldsymbol{Q}'_a$ under the* rate-wise stochastic ordering, *if:*

1. $\boldsymbol{P}_a \leq_{\mathrm{st}} \boldsymbol{P}'_a$

2. *For all states s:*

   (a) If $r_a(s) > 0: \; 1 \leq \dfrac{r'_a(s)}{r_a(s)} \leq \min\limits_{s' < s} \left\{ \dfrac{1 - \sum_{t > s'} \boldsymbol{P}_a(s,t)}{1 - \sum_{t > s'} \boldsymbol{P}'_a(s,t)} \right\}$

   (b) If $r_a(s) = 0 \wedge r'_a(s) > 0: \; \forall s' < s. \; \sum_{t > s'} \boldsymbol{P}'_a(s,t) = 1$

Definition 6.4.4. *A generator matrix component $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ is* rate-wise monotone *if:*

1. *$\boldsymbol{P}_a$ is monotone.*

2. *For all states s, s' such that $s \prec s'$:*

   (a) If $r_a(s) > 0: \; 1 \leq \dfrac{r_a(s')}{r_a(s)} \leq \min\limits_{s'' < s} \left\{ \dfrac{1 - \sum_{t > s''} \boldsymbol{P}_a(s,t)}{1 - \sum_{t > s''} \boldsymbol{P}_a(s',t)} \right\}$

   (b) If $r_a(s) = 0 \wedge r_a(s') > 0: \; \forall s'' < s. \; \sum_{t > s''} \boldsymbol{P}_a(s',t) = 1$

Intuitively, in both cases, we ensure that the probability transition matrix increases faster than the rate function, so that after uniformisation we remain monotone and comparable. Note that the original generator matrix component is not necessarily rate-wise monotone — monotonicity is only required of the upper bound that we construct.

It is important to comment on the above definitions in the case when $r_a(s) = 0$. If the numerator ($r'_a(s)$ or $r_a(s')$) is also zero, then there is no additional condition on the probabilities, as the rate does not increase. If the numerator is greater than zero, however, the ratio of the rates is undefined, and so we instead ensure that the denominator on the right side of the inequality is zero — i.e. the ratio of the probabilities is also undefined. In real terms, this means that we are effectively blocked from adding probability mass below the diagonal element for previously disabled activities, potentially leading to a looser bound than if the activity had been enabled.

We can show that the strong stochastic ordering and monotonicity follow from rate-wise stochastic comparison and monotonicity. This means that any CTMC that we construct by this method is stochastically comparable in the usual sense. We state this in the following two theorems:

Theorem 6.4.5. *If $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I}) \leq_{\mathrm{rst}} \boldsymbol{Q}'_a = r'_a(\boldsymbol{P}'_a - \boldsymbol{I}')$ and for all $s \in S$, $r_a(s) \leq r'_a(s)$, then $\boldsymbol{Q}_a \leq_{\mathrm{st}} \boldsymbol{Q}'_a$.*

Theorem 6.4.6. *If $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ is rate-wise monotone, and for all $s \prec s' \in S$, $r_a(s) \leq r_a(s')$, then $\boldsymbol{Q}_a$ is monotone.*

Unfortunately, it is still not the case that rate-wise monotonicity and rate-wise stochastic ordering are preserved in general when two components cooperate. The problem arises because of the minimum operator, which is applied to the rate functions. If we take monotonicity, for example, the probabilistic transition matrix is constrained according to the ratio between successive rates. When we compose two monotone components, however, it is possible for one to be completely bounded by the other in terms of its ability to perform an activity of type a. That is to say, the rate of performing a in each state of one component might be less than the rate of a in *any* state of the other. Hence the minimum of the two rate functions, and the resulting constraint on the composed probabilistic transition matrix, depends on only one of the components. The required constraint on the composed matrix may therefore be tighter than that for one of the components, and so the rate-wise monotonicity and ordering may fail to hold.

This problem is more apparent if we look at a particular example:

$$r_a(\boldsymbol{P}_a - \boldsymbol{I}) = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix} \left(\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} - \boldsymbol{I} \right)$$

Note that this is similar to Equation 6.9, which illustrated why we need rate-wise comparison and monotonicity. In this case, the component *is* rate-wise monotone, but if it were to synchronise with a component that has a rate function of, say, $[1,1,2]$, we would arrive at the same problem as before.

It is therefore not possible for us to construct a bound for a sequential component, without considering the *context* in which it occurs. To define this context, we need a measure on components, to indicate the extent to which the rate function increases. For monotonicity, we are concerned with the ratio between successive rates, and in particular the maximum of these. This is because, when taking the Kronecker product, we consider all possible state combinations. Hence the maximum increase *will* actually occur, and gives a bound on how a component can affect those that it cooperates with.

Definition 6.4.7. *The* internal rate measure *of a component C, with generator matrix $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ for action type a, is:*

$$\|C\|_a = \begin{cases} \top & \text{if } \exists s.\ \mathrm{succ}(s) \neq \emptyset \wedge r_a(s) = 0 \\ \max\limits_{s,s'} \left\{ \dfrac{r_a(s')}{r_a(s)} \;\middle|\; s' \in \mathrm{succ}(s) \right\} & \text{otherwise} \end{cases}$$

Note that $\max\emptyset = 0$. Here, $\mathrm{succ}(s)$ denotes the set of immediate successors of the state s as defined by the simple partial order[6]. More precisely, we say that $s' \in \mathrm{succ}(s)$ if $s' > s \wedge \neg\exists s''.\ s' > s'' > s$. In the case of stochastic ordering, we need to compare the rate functions of two components (the original and the bound), but otherwise the same principle applies:

Definition 6.4.8. *The* comparative rate measure *of components C and C', with generator matrices $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ and $\boldsymbol{Q}'_a = r'_a(\boldsymbol{P}'_a - \boldsymbol{I})$ respectively for action type a, is defined as:*

$$\left\|C, C'\right\|_a = \begin{cases} \top & \text{if } \exists s.\ \mathrm{succ}(s) \neq \emptyset \wedge r_a(s) = 0 \\ \max\limits_{s} \left\{ \dfrac{r'_a(s)}{r_a(s)} \right\} & \text{otherwise} \end{cases}$$

In the above definitions, note that the ratio may be undefined (i.e. $r_a(s) = 0$ for some s). In this case, we define the rate measure to have the value $\top$, which dominates all the real numbers.

We can now state precisely what we mean by a *context*, which is slightly different to a conventional process algebra definition, since we care only about those components that can affect the rate at which we perform an activity.

Definition 6.4.9. *The context © of a component C is the set of all components that it can cooperate with, as defined by the system equation. We say that © is* internally

[6] Note that this successor function comes from our partial order of the state space (i.e. the order that tells us which probabilities we can compare), and not from the transition relation of the Markov chain.

bounded *by* $B_{int} \in \mathbb{R}_{\geq 0} \cup \{\top\}$*, with respect to action type a, if:*

$$\forall C_i \in ©.\ \|C_i\|_a \leq B_{int}$$

Furthermore, © *and* ©′ *are* comparatively bounded *by* $B_{comp} \in \mathbb{R}_{\geq 0} \cup \{\top\}$*, with respect to action type a, if:*

$$\forall C_i \in ©, C_i' \in ©'.\ \left\|C_i, C_i'\right\|_a \leq B_{comp}$$

Note that '©' should be read as an atomic symbol, having no relation to 'C' as used for a component. Since the internal and comparative bounds depend only on the rate functions, we have a simple algorithm for computing them. If we construct a monotone upper bound of each rate function before bounding the transition matrices, then the internal bound of a context, for example, is simply the maximum of the internal rate measures of the components within the context.

This leads us to the final extension of our definitions — the *context-bounded rate-wise stochastic ordering* and *context-bounded rate-wise monotonicity*, which extend Definitions 6.4.3 and 6.4.4 respectively. Intuitively, they require the rate function $r_{i,a}$ of component i to not increase faster than is allowed for by the matrices $\boldsymbol{P}_{j,a}$, $j \neq i$ of all the components it cooperates with.

Definition 6.4.10. *Given two generator matrix components* $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ *and* $\boldsymbol{Q}_a' = r_a'(\boldsymbol{P}_a' - \boldsymbol{I})$*, we say that* $\boldsymbol{Q}_a \leq_{\text{rst}}^{B_{comp}} \boldsymbol{Q}_a'$ *under the context-bounded rate-wise stochastic ordering, if:*

1. $\boldsymbol{P}_a \leq_{\text{st}} \boldsymbol{P}_a'$

2. *For all states s:*

   (a) *If* $r_a(s) > 0 \wedge B_{comp} \neq \top :\ 1 \leq \max\left\{\frac{r_a'(s)}{r_a(s)}, B_{comp}\right\} \leq \min_{s'<s}\left\{\frac{1 - \sum_{t>s'} \boldsymbol{P}_a(s,t)}{1 - \sum_{t>s'} \boldsymbol{P}_a'(s,t)}\right\}$

   (b) *If* $(r_a(s) = 0 \wedge r_a'(s) > 0) \vee B_{comp} = \top :\ \forall s' < s.\ \sum_{t>s'} \boldsymbol{P}_a'(s,t) = 1$

We can extend the definition of rate-wise monotonicity similarly:

Definition 6.4.11. *A generator matrix component* $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ *is context-bounded rate-wise monotone with respect to* B_{int}*, if:*

1. $\boldsymbol{P}_a$ *is monotone.*

2. *For all states s, s' such that $s \prec s'$:*

   (a) *If* $r_a(s) > 0 \wedge B_{int} \neq \top :\ 1 \leq \max\left\{\dfrac{r_a(s')}{r_a(s)}, B_{int}\right\} \leq \min_{s'' \prec s}\left\{\dfrac{1 - \sum_{t \succ s''} \boldsymbol{P}_a(s,t)}{1 - \sum_{t \succ s''} \boldsymbol{P}_a(s',t)}\right\}$

   (b) *If* $(r_a(s) = 0 \wedge r_a(s') > 0) \vee B_{comp} = \top :\ \forall s'' \prec s.\ \sum_{t \succ s''} \boldsymbol{P}_a(s',t) = 1$

We can now prove that the CTMC of the system, after composing the individually-bounded generator matrix components, is a monotone, lumpable, upper bound of the concrete CTMC, with respect to the ordering on each component. Recall the shorthand $\circledast$, which was defined as:

$$(r_1, \boldsymbol{P}_1) \circledast (r_2, \boldsymbol{P}_2) = \min\{r_1, r_2\}(\boldsymbol{P}_1 \otimes \boldsymbol{P}_2 - \boldsymbol{I}_{|S_1|} \otimes \boldsymbol{I}_{|S_2|})$$

where S_1 and S_2 are the state spaces of components C_1 and C_2 respectively.

If these state spaces are partially ordered according to $(S_1, \prec_1)$ and $(S_2, \prec_2)$ respectively, the $\circledast$ operator preserves stochastic comparison and monotonicity with respect to the lifted orders $(S_1 \times S_2, \prec_1^L)$ and $(S_1 \times S_2, \prec_2^L)$. We say $(s_1, s_2) \prec_1^L (s_1', s_2')$ if $s_1 \prec_1 s_1'$, and $\neg\exists s_2''.\ s_2' \prec_2 s_2''$. In other words, s_1 and s_1' must be comparable, but there are no constraints on s_2 and s_2' other than that there are no states above s_2' in the ordering $\prec_2$. $\prec_2^L$ is defined similarly.

The intuition behind these lifted orders is that we can only compare probabilities of the entire PEPA model if we project back down onto the state space of one component. In other words, we are able to bound steady state probability of a component being in a certain state, in the context of the rest of the model. We would prefer to have a stronger order than this, such as the product order, but unfortunately this does not follow from the context-bounded rate-wise monotonicity and ordering. We will discuss the advantages and limitations of our approach further in Chapter 8.

Theorem 6.4.12 (Monotonicity). *Let two components, C_1 and C_2, occur in contexts $\copyright_1$ and $\copyright_2$ respectively, where $C_1 \in \copyright_2$ and $C_2 \in \copyright_1$. Let $\copyright_1$ be internally bounded by B_{int}^1 and $\copyright_2$ by B_{int}^2, for action type a.*

If the matrices $\boldsymbol{Q}_{1,a} = r_{1,a}(\boldsymbol{P}_{1,a} - \boldsymbol{I})$ of C_1 and $\boldsymbol{Q}_{2,a} = r_{2,a}(\boldsymbol{P}_{2,a} - \boldsymbol{I})$ of C_2 are context-bounded rate-wise monotone by B_{int}^1 and B_{int}^2 respectively, then $(r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a})$ is context-bounded rate-wise monotone by the internal bound B_{int}^3 of the context $\copyright_1 \cap \copyright_2$ of $C_1 \underset{L}{\bowtie} C_2$, for all action sets L.

Theorem 6.4.13 (Lumpability). *Let C_1 and C_2 be PEPA models with generator matrices $\boldsymbol{Q}_1 = \sum_a \boldsymbol{Q}_{1,a}$ and $\boldsymbol{Q}_2 = \sum_a \boldsymbol{Q}_{2,a}$, where $\boldsymbol{Q}_{1,a} = r_{1,a}(\boldsymbol{P}_{1,a} - \boldsymbol{I})$ and $\boldsymbol{Q}_{2,a} = r_{2,a}(\boldsymbol{P}'_{2,a} - \boldsymbol{I})$. Then for all action types a, if the terms $\boldsymbol{Q}_{1,a}$ in $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_{2,a}$ in $\boldsymbol{Q}_2$ are ordinarily lumpable according to the partitions $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively, then the term $\boldsymbol{Q}_a = (r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a})$ in $\boldsymbol{Q} = \sum_a \boldsymbol{Q}_a$ is ordinarily lumpable according to $\mathcal{L}_1 \times \mathcal{L}_2$.*

Theorem 6.4.14 (Stochastic Order). *Consider the components C_i and C'_i, with generator matrices $\boldsymbol{Q}_{i,a} = r_{i,a}(\boldsymbol{P}_{i,a} - \boldsymbol{I})$ and $\boldsymbol{Q}'_{i,a} = r'_{i,a}(\boldsymbol{P}'_{i,a} - \boldsymbol{I})$, for $i \in \{1,2\}$ and action type a. Let $\boldsymbol{Q}_{i,a} \leq_{\mathrm{rst}}^{B^i_{comp}} \boldsymbol{Q}'_{i,a}$, with contexts $©_i \leq_{\mathrm{st}} ©'_i$, where B^i_{comp} is the comparative bound of $©_i$ and $©'_i$. If B^3_{comp} is the comparative bound of the contexts $©_1 \cap ©_2$ and $©'_1 \cap ©'_2$, we have $(r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a}) \leq_{\mathrm{rst}}^{B^3_{comp}} (r'_{1,a}, \boldsymbol{P}'_{1,a}) \circledast (r'_{2,a}, \boldsymbol{P}'_{2,a})$.*

The preservation of monotonicity and stochastic order is ensured by the definitions we have developed. Lumpability is preserved by the PEPA cooperation combinator, as a consequence of the strong equivalence congruence [91]. Equivalent theorems also hold for the compositional addition operator $(r_{1,a}, \boldsymbol{P}_{1,a}) + (r_{2,a}, \boldsymbol{P}_{2,a})$ — we do not state the theorems here, and their proofs follow the same pattern as for the above. Since the $\odot$ operator is defined in terms of $\circledast$ and compositional addition, this ensures that both $\circledast$ and $\odot$ preserve context-bounded rate-wise monotonicity and stochastic ordering, and lumpability.

So that we can expand out the Kronecker form given in Definition 6.1.2 by adding together the generator matrices for different action types, we also need to ensure that normal matrix addition preserves the stochastic ordering and monotonicity. Note that we only require the normal stochastic ordering and monotonicity at this stage, because we no longer need to compose these generator matrices together. The following two theorems hold for matrices over the same partially-ordered state space $(S, \prec)$:

Theorem 6.4.15. *If $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I}) \leq_{\mathrm{rst}} \boldsymbol{Q}'_a = r'_a(\boldsymbol{P}'_a - \boldsymbol{I})$ and for all $s \in S$, $r_a(s) \leq r'_a(s)$, and if $\boldsymbol{Q}_b = r_b(\boldsymbol{P}_b - \boldsymbol{I}) \leq_{\mathrm{rst}} \boldsymbol{Q}'_b = r'_b(\boldsymbol{P}'_b - \boldsymbol{I})$ and for all $s \in S$, $r_b(s) \leq r'_b(s)$, then $\boldsymbol{Q}_a + \boldsymbol{Q}_b \leq_{\mathrm{st}} \boldsymbol{Q}'_a + \boldsymbol{Q}'_b$.*

Theorem 6.4.16. *If $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ is rate-wise monotone, and for all $s \prec s' \in S$, $r_a(s) \leq r_a(s')$, and if $\boldsymbol{Q}_b = r_b(\boldsymbol{P}_b - \boldsymbol{I})$ is rate-wise monotone, and for all $s \prec s' \in S$, $r_b(s) \leq r_b(s')$, then $\boldsymbol{Q}_a + \boldsymbol{Q}_b$ is monotone.*

A consequence of the above theorems is that by constructing an upper bound for the rate function and probability transition matrix of each sequential component for each action type, we get an upper bound for the CTMC of the entire model when we multiply out the Kronecker form.

6.4.2 An Algorithm for Bounding PEPA Components

In this section, we will describe an algorithm for constructing an upper bound for a steady state property of a PEPA model. This is an extension of the algorithm by Fourneau *et al.* that we introduced in Section 5.6.2, in that it constructs, compositionally, context-bounded rate-wise upper-bounding rate functions and probabilistic transition matrices for the components in a PEPA model. It also uses a *partially ordered* state space, rather than a total order — we present the algorithm here in the context of the simple partial order from Definition 6.4.1.

Our algorithm is as follows. We take a PEPA model of the form $C_1 \underset{L_1}{\bowtie} \cdots \underset{L_{n-1}}{\bowtie} C_n$. For each action type $a \notin L_1 \cup \ldots \cup L_{n-1}$, we rename a to the local action type τ — i.e. we group all local transitions together. Each component C_i has an abstraction $(S^\sharp_i, \alpha_i)$, and a simple partial order specified by $\mathcal{B}_i = \{\mathcal{B}_{i,1}, \ldots \mathcal{B}_{i,m_i}\}$.

Algorithm 3 An algorithm for constructing a context-bounded rate-wise upper-bounding probability transition matrix

```
$y' \leftarrow |S^\sharp_i|$
for $y \leftarrow |S^\sharp_i|$ to 1 do
  if $b(y) = b(\mathcal{B}_k)$ for some $\mathcal{B}_k$ then
    $refresh\_sum(\boldsymbol{P}, \boldsymbol{R}, b(y), e(y'))$
    $normalise(\boldsymbol{R}, b(y), e(y'))$
    for $p \leftarrow y'$ to $y$ do
      $normalise\_partition(\boldsymbol{R}, b(p), e(p))$
    end for
    $y' \leftarrow y - 1$
  end if
end for
```

1. For each component C_i, we construct a mapping $\mathcal{I}_i$ from its state space S_i to matrix indices $\{1, \ldots, |S_i|\}$, so that states in the same partition have contiguous indices, which are ordered such that $s \prec s' \Rightarrow \mathcal{I}_i(s) < \mathcal{I}_i(s')$.

2. We compute a lumpable monotone upper-bounding rate function $r'_{i,a}$ from the rate function $r_{i,a}$ of each component C_i and action type $a \neq \tau$:

$$r'_{i,a}(s \in \mathcal{B}_{i,k}) = \max\left(\bigcup_{j=k+1}^{m_i} \{ r_{i,a}(s') \mid s' \in \mathcal{B}_{i,j} \} \cup \{ r_{i,a}(s') \mid \alpha_i(s') = \alpha_i(s) \} \right)$$

3. We calculate the internal bound B_{int} and comparative bound B_{comp} for the context of each component and action type $a \neq \tau$, using the bounded rate functions.

4. We compute an upper-bounding probability transition matrix $\boldsymbol{R}_{i,a}$ from the transition matrix $\boldsymbol{P}_{i,a}$ of each component C_i and action type $a \neq \tau$ (Algorithm 3).

5. For the internal action type τ, we uniformise the generator matrix $\boldsymbol{Q}_{i,\tau}$ of each component C_i, and apply Algorithm 3 with no context constraints. Since uniformisation ensures that every state has the same exit rate, the rate-wise ordering and monotonicity constraints reduce to the standard stochastic ordering and monotonicity — hence we can obtain tighter bounds.

6. We construct and solve the generator matrix obtained by multiplying out the Kronecker representation of the upper-bounding model[7].

It is important to note that not all of the components in the model necessarily need to have ordering constraints on their state space. For example, if we are interested in a property of just one component — i.e. the projection from the state space of the system onto that of the component — then we have no particular constraints on the probability distributions of the other components. But what does this mean in terms of constructing a bound for that component? The theorems in the previous section only account for when we *need* to bound a component. If we wish to exclude one of the components, we have to assume the 'worst' case — that is to say, that the component does not have any effect on the rest of the system.

Intuitively, when we bound a component, we maximise the probability of moving into higher valued states in the ordering. Since cooperation in PEPA takes the minimum of two rates, it is possible for a component to limit, but not increase, the transition rates for a particular action type. Hence a *monotone* upper bound for a component is a true upper bound in the worst case context. This means that we can ignore other components and still obtain, locally, an upper bound.

Let us examine Algorithm 3 in more detail. This takes as input a probability transition matrix $\boldsymbol{P}$, and an empty matrix $\boldsymbol{R}$ (of the same dimensions) in which to construct the monotone and lumpable upper bound. We assume that the upper bound r' of the rate function r has already been constructed, along with the internal and comparative

[7] To implement this, we explore the transition system generated by the new model, rather than performing the Kronecker multiplications explicitly. This avoids including unreachable states, which would result in a singular generator matrix.

Algorithm 4 *refresh_sum*$(\boldsymbol{P}, \boldsymbol{R}, b, e)$

```
for $\mathcal{B}_k \leftarrow \mathcal{B}_1$ to $\mathcal{B}_m$ do
    $R_{max} \leftarrow \max_{s \in \mathcal{B}_{k-1}} \sum_{j=b}^{|S|} \boldsymbol{R}(s, j)$
    $P_{max} \leftarrow \max_{s \in \mathcal{B}_k} \sum_{j=b}^{|S|} \boldsymbol{P}(s, j)$
    $B_I \leftarrow 1 - \min\left\{ B_{int}, \frac{\max_{s \in \mathcal{B}_k} r(s)}{\max_{s \in \mathcal{B}_k} r'(s)} \right\}(1 - R_{max})$
    $B_C \leftarrow 1 - \min\left\{ B_{comp}, \frac{\max_{s \in \mathcal{B}_{k-1}} r'(s)}{\max_{s \in \mathcal{B}_k} r'(s)} \right\}(1 - P_{max})$
    for $i \leftarrow b(\mathcal{B}_k)$ to $e(\mathcal{B}_k)$ do
        if $i \geq b$ then
            $\Sigma_{new} \leftarrow \max\{R_{max}, P_{max}, B_I, B_C\}$
        else
            $\Sigma_{new} \leftarrow \max\{R_{max}, P_{max}\}$
        end if
        $P_{new} \leftarrow \Sigma_{new} - \sum_{j'=e+1}^{|S|} \boldsymbol{R}(i, j')$
        $P_{old} \leftarrow \sum_{j=b}^{e} \boldsymbol{P}(i, j)$
        for $j \leftarrow b$ to $e$ do
            if $P_{old} > 0$ then
                $\boldsymbol{R}(i, j) \leftarrow \frac{\boldsymbol{P}(i,j)}{P_{old}} P_{new}$
            else
                $\boldsymbol{R}(i, j) \leftarrow \frac{1}{e-b+1} P_{new}$
            end if
        end for
    end for
end for
```

bounds, B_{int} and B_{comp}. We define $b(y)$ and $e(y)$ respectively as the minimum and maximum index in the set $\{\mathcal{I}(s) \mid \mathcal{I}^{\sharp}(\alpha(s)) = y\}$[8]. $b(\mathcal{B}_k)$ and $e(\mathcal{B}_k)$ are defined similarly for the set $\{\mathcal{I}(s) \mid s \in \mathcal{B}_k\}$.

Algorithm 5 *normalise*($\boldsymbol{R}, b, e$)

```
for $y \leftarrow 1$ to $|S^{\sharp}|$ do
    $R_{new} \leftarrow \max_{i=b(y)}^{e(y)} \sum_{j=b}^{e} \boldsymbol{R}(i,j)$
    for $i \leftarrow b(y)$ to $e(y)$ do
        $R_{old} \leftarrow \sum_{j=b}^{e} \boldsymbol{R}(i,j)$
        for $j \leftarrow b$ to $e$ do
            if $R_{old} > 0$ then
                $\boldsymbol{R}(i,j) \leftarrow \frac{\boldsymbol{R}(i,j)}{R_{old}} R_{new}$
            else
                $\boldsymbol{R}(i,j) \leftarrow \frac{1}{e-b+1} R_{new}$
            end if
        end for
    end for
end for
```

Algorithm 6 *normalise_partition*($\boldsymbol{R}, b, e$)

```
for $y \leftarrow 1$ to $|S^{\sharp}|$ do
    $R_{average} \leftarrow \frac{1}{e(y)-b(y)+1} \sum_{i=b(y)}^{e(y)} \sum_{j=b}^{e} \boldsymbol{R}(i,j)$
    for $i \leftarrow b(y)$ to $e(y)$ do
        for $j \leftarrow b$ to $e$ do
            $\boldsymbol{R}(i,j) \leftarrow R_{average}$
        end for
    end for
end for
```

Algorithm 3 makes use of three sub-procedures:

1. *refresh_sum*($\boldsymbol{P}, \boldsymbol{R}, b, e$) (Algorithm 4) ensures that for each ordering block, from indices b to e, the matrix $\boldsymbol{R}$ is context-bounded rate-wise monotone, and an up-

[8] $\mathcal{I}^{\sharp}$ is defined such that $\mathcal{I}(s) < \mathcal{I}(s') \Rightarrow \mathcal{I}^{\sharp}(\alpha(s)) \leq \mathcal{I}^{\sharp}(\alpha(s'))$.

per bound of $\boldsymbol{P}$. The core of this algorithm is the computation of Σ_{new}, where the bounds B_I and B_C come directly from re-arranging the definitions of context-bounded rate-wise stochastic ordering and monotonicity respectively (Definitions 6.4.10 and 6.4.11). These additional constraints are only needed for elements on or below the diagonal ($i \geq b$), since the $-r_a\boldsymbol{I}$ term — which is the reason for the rate-wise extension to stochastic ordering and monotonicity — only applies to sums that include the diagonal element.

To achieve a new row sum of Σ_{new}, we adjust the individual entries in $\boldsymbol{R}$ so that the relative probabilities are preserved. This is a choice that we make, to minimise our impact on the matrix — because we do not have a total order, we can distribute the probability mass within an ordering block in any way.

2. *normalise*$(\boldsymbol{R}, b, e)$ (Algorithm 5) ensures that for each partition in the ordering block from indices b to e, states in the same partition have the same probability of moving to a different ordering block.

3. *normalise_partition*$(\boldsymbol{R}, b, e)$ (Algorithm 6) ensures that each state in the partition from indices b to e has the same probability of moving to another partition. We choose to assign the average transition probabilities.

Essentially, the *normalise* procedure ensures lumpability of ordering blocks — by 'borrowing' probability mass from lower ordering blocks — which preserves monotonicity. The *normalise_partition* procedure then ensures lumpability of partitions, by re-distributing probability mass within the same ordering block.

We can compare the time complexity of our algorithm to that of Fourneau [69] from the previous chapter:

Property 6.4.17. *The worst case time complexity for Algorithm 3 is* $O(|S|^2)$*, where S is the state space of the component we apply the algorithm to.*

This follows because the *refresh_sum*, *normalise*, and *normalise_partition* procedures each contribute $O(|S|^2)$ operations in the worst case. In this sense, the additional loop around *normalise_partition* in Algorithm 3 is misleading. For each ordering block, we apply *normalise* to the block, and *normalise_partition* to each partition within the block — since the number of states in all the partitions within a block is equal to the number of states within the block, these two procedures result in the same number of operations. Similarly, note that in the *normalise* algorithm, whilst there are three nested loops, the outermost loop iterates over all the partitions (abstract states), and the second loop iterates over all the states within a partition — this is consequently the same as a single loop that iterates over all the concrete states in S.

Recall that the complexity of Fourneau's algorithm was also $O(|S|^2)$, and so we can see that compositionality and partial ordering do not affect the worst case time complexity, except by a constant factor. We should note, however, that since our algorithm is applied compositionally, the state space S is only that of an individual component. This means that overall, the complexity can be much better than a direct application of Fourneau's algorithm. If we have n components, each with a state space S, then Fourneau's algorithm has a worst case time complexity of $O(|S|^{2n})$, whereas the compositional algorithm has complexity $O(n|S|^2A)$, where A is the number of distinct action types in the PEPA model. Of course, this does not take into account the cost of expanding the compositional form, but recall that we only expand the lumped Markov chain, which is much smaller than the original.

6.4.3 An Example of Stochastic Bounding

As an example, let us return to the PEPA model in Figure 6.1. The generator matrix has the following Kronecker form, from Equation 6.6, which we repeat below, except that instead of expanding out the $\odot$ operator for the independent action type τ, we express this directly as a Kronecker sum on the generator matrix terms. This is entirely equivalent, due to Theorem 6.1.1, since the action type is independent throughout the system equation. We can therefore construct a normal stochastic bound for τ actions, which will be more precise than in the context-bounded rate-wise case.

$$
\begin{aligned}
\boldsymbol{Q} = & \left(\begin{bmatrix} 0 & 0 & 0 \\ r_2 & -2r_2 & r_2 \\ r_3 & r_3 & -2r_3 \end{bmatrix} \oplus \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \right) \\
& + \min\left\{ \begin{bmatrix} r_a \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_D \\ 0 \end{bmatrix} \right\} \left(\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \\
& + \min\left\{ \begin{bmatrix} r_b \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ r_D \end{bmatrix} \right\} \left(\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)
\end{aligned}
$$

Suppose that we want to aggregate states C_2 and C_3 of the first component, and that we also want to compute an upper bound of being in either state C_2 or C_3, and in state

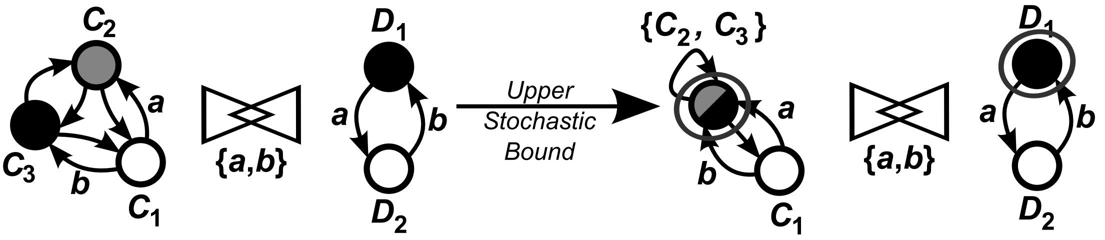


Figure 6.6: Stochastic upper bound of a PEPA model

D_1, in the steady state. This is illustrated in Figure 6.6. We will choose the rates to be as follows:

$$\begin{aligned} r_a = r_2 = r_D &= 1 \\ r_b = r_3 &= 2 \end{aligned}$$

We can now apply the algorithm we presented, to construct the following upper-bounding model:

$$\begin{aligned}
\boldsymbol{Q}_U = & \left(4 \left(\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} - \boldsymbol{I} \right) \oplus \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \right) \\
+ & \min \left\{ \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\} \left(\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \\
+ & \min \left\{ \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\} \left(\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)
\end{aligned}$$

Note that to bound the D component, we should really permute the rows of the matrix so that state D_1 (our top state) has the highest index — this is not the case here. In isolation, D is already rate-wise monotone, but because it must also be *context-bounded*, and it is possible for C to perform or not to perform an a activity, we are forced to loosen the probability transition matrix for a in the above above. The upper bound for the internal probability transition matrix is computed using the standard algorithm of Fourneau *et al*, as described in Section 5.6.2, since these activities are entirely independent of any other component. We can now construct the aggregated model as follows:

$$\begin{aligned}\boldsymbol{Q}^{\sharp}_{U} = {} & \left(4\left(\begin{bmatrix} 1 & 0 \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix} - \boldsymbol{I}\right) \oplus \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\right) \\ & + \min\left\{\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right\}\left(\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\ & + \min\left\{\begin{bmatrix} 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right\}\left(\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)\end{aligned}$$

Multiplying this out, we have a lumped upper bound for the generator matrix $\boldsymbol{Q}^{\sharp}$:

$$\boldsymbol{Q}^{\sharp}_{U} = \begin{array}{c} (C_1D_1) \\ (C_1D_2) \\ (C_{\{2,3\}}D_1) \\ (C_{\{2,3\}}D_2) \end{array} \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -2 & 1 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & -2 \end{bmatrix}$$

For clarity, the rows of the above matrix are labelled with the states they correspond to. Notice that the bound is quite coarse, as it results in a bottom strongly-connected component corresponding to component D being in state D_1. If we solve this CTMC, we find that the probability of component C being in state $C_{\{2,3\}}$ is 0.5. This is an upper bound of the probability of component C being in state C_2 or C_3 in the original Markov chain, which is 0.42857. The probability of component D being in state D_1 is 1, which is an upper bound of the actual probability in the original Markov chain of 0.47619. In this small example, the bounds are not particularly tight. In the next chapter, however, we will see an example of a larger model for which much better bounds can be obtained, using our tool.

6.5 Summary of Results

In this chapter, we have seen how two techniques for computing bounding abstractions of Markov chains can be applied compositionally to PEPA models. Abstract Markov chains are used for model checking CSL/X properties, excluding the steady state operator, and stochastic bounds are used to bound steady state probabilities. By combining these techniques we are able to model check all CSL/X properties. Whilst we lose some precision by performing the abstractions compositionally, it has the advantage of allowing us to deal with much larger models. In particular, if the state space of the model is too large to store, we cannot apply the abstraction at the level of the Markov chain, and so a compositional abstraction is the only viable approach.

It should be noted that while the results in this chapter are specific to PEPA, the same techniques can be applied to other stochasic process algebras with relatively minor modifications. For example, in EMPA [28] only active-passive synchronisation can occur, and so we should be able to deal with it using a simplification of our construction for PEPA. In TIPP [79], synchronisation results in a *multiplication* of the rates (as opposed to taking the minimum), and so we would have to adapt our construction. However, we would expect to obtain properties such as monotonicity more easily in this case — recall that it was the minimum operator in PEPA that gave us the greatest difficulties when constructing compositional stochastic bounds. There has already been work on compositional abstraction of IMC [87] using modal transitions (essentially, an MDP-based abstraction) [104], and our abstract Markov chain technique could also be applied.

The work in this chapter addresses an important problem, from both the perspective of performance-driven development and performance modelling in general. Being able to analyse extremely large models — such as the ones we extract from program code — is essential if we are to build a useful framework for performance-driven development. Most importantly, we need *tool support* if techniques such as those in this chapter are to be used in practice. In the next chapter we will describe a tool that we have developed on top of the Eclipse platform [1], which provides both a graphical interface for our abstraction techniques, and a model checker for analysing abstracted PEPA models.

Chapter 7

A Stochastic Model Checker for PEPA

In the previous two chapters, we have examined various methods that allow us to analyse large Markov chains and PEPA models, using state space abstractions. In particular, we introduced two techniques — abstract Markov chains and stochastic bounds — that together allow us to compositionally abstract PEPA models so that we can model check CSL/X properties. To use these techniques in practice, however, it is vital that we have tool support — and in particular, that we hide the details of our abstractions away from the user as much as possible. To this end, we will describe in this chapter our tool for abstracting and model checking PEPA models.

The PEPA modelling language has a long history of tool support, supporting a variety of different analyses. These include the PEPA Workbench [78] for steady state state analysis, the Imperial PEPA Compiler (IPC) [34] for passage time analysis, and the PRISM model checker [114]. Recently, efforts have been made to unify the tools under a common interface, leading to the PEPA plug-in for Eclipse [178]. This supports both steady state Markovian analysis and time series analysis — using both the ordinary differential equation semantics of PEPA [92] and stochastic simulation [33].

Currently, the only model checker that directly supports PEPA as an input language is PRISM [114, 96], although it can only handle models that perform active-passive synchronisation. The main input language for PRISM is based on a Markov Decision Process (MDP) [146], and it also supports model checking of DTMCs and CTMCs. The Markov Reward Model Checker (MRMC) [106] also supports Markov reward models, and uniform Continuous Time MDPs (CTMDPs). In addition, it uses abstract CTMCs (ACTMCs) directly as an abstraction — to perform model checking, note

$$\begin{aligned}
P_{14} &\overset{def}{=} (reg_{14}, r).P_{14} + (move_{15}, m).P_{15} \\
P_{15} &\overset{def}{=} (reg_{15}, r).P_{15} + (move_{14}, m).P_{14} + (move_{16}, m).P_{16} \\
P_{16} &\overset{def}{=} (reg_{16}, r).P_{16} + (move_{15}, m).P_{15}
\end{aligned}$$

$$\begin{aligned}
S_{14} &\overset{def}{=} (reg_{14}, \top).(rep_{14}, s).S_{14} \\
S_{15} &\overset{def}{=} (reg_{15}, \top).(rep_{15}, s).S_{15} \\
S_{16} &\overset{def}{=} (reg_{16}, \top).(rep_{16}, s).S_{16}
\end{aligned}$$

$$\begin{aligned}
DB_{14} &\overset{def}{=} (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} + (rep_{16}, \top).DB_{16} \\
DB_{15} &\overset{def}{=} (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} + (rep_{16}, \top).DB_{16} \\
DB_{16} &\overset{def}{=} (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} + (rep_{16}, \top).DB_{16}
\end{aligned}$$

$$P_{14} \underset{\{reg_{14}, reg_{15}, reg_{16}\}}{\bowtie} (S_{14} \parallel S_{15} \parallel S_{16}) \underset{\{rep_{14}, rep_{15}, rep_{16}\}}{\bowtie} DB_{14}$$

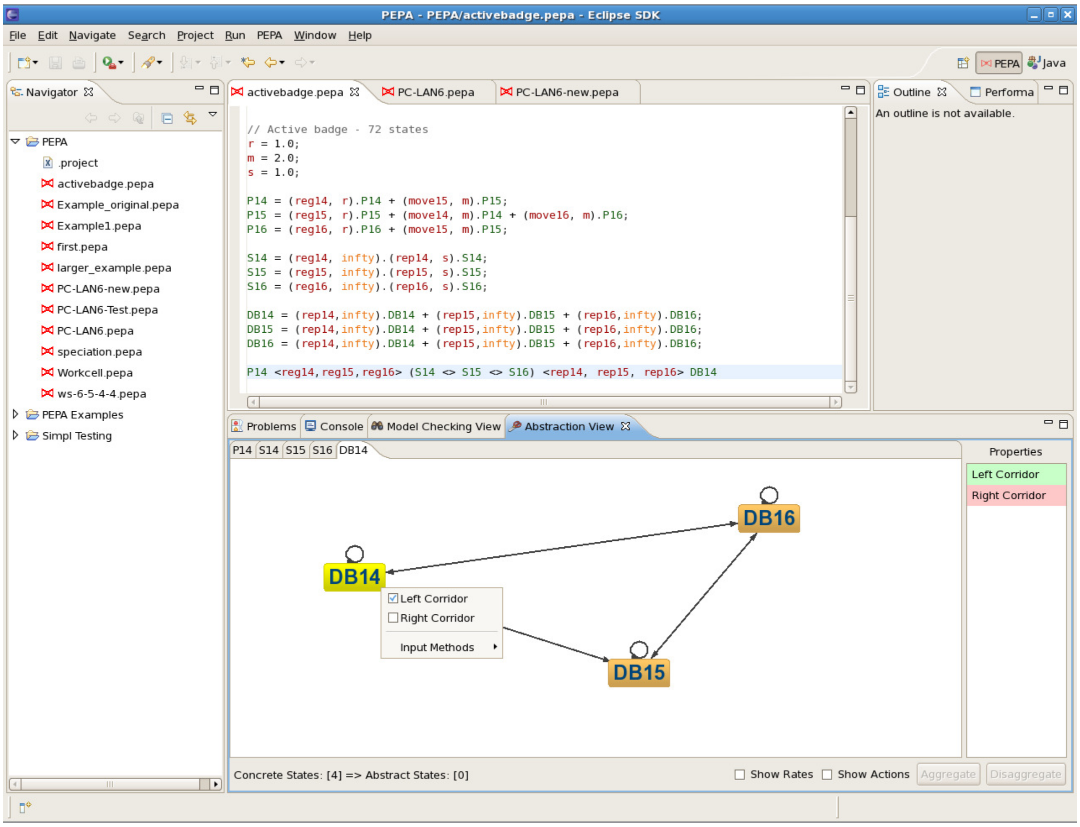
Figure 7.1: A PEPA model of an active badge system

that an ACTMC can be converted into a CTMDP with finite branching if we only consider the extreme distributions, and this satisfies the same CSL formulae as the ACTMC [105].

To implement the compositional abstraction techniques that we developed in Chapter 6, we decided to directly extend the PEPA plug-in for Eclipse. This has the advantage of flexibility, since we can completely hide from the user the internal details of the abstractions used. The cost of building a stand-alone tool is that we had to implement our own model checker, but this allows us more flexibility in the algorithms we use. Our tool is the first model checker that natively supports the minimum semantics of PEPA cooperation, and that performs a compositional abstraction directly on PEPA models. There is no reason why we cannot also interface to the other model checkers in the future, and it would be straightforward to output to MRMC, for example.

In this chapter, we will describe our extensions to the PEPA plug-in for Eclipse, which provide a graphical interface for abstracting PEPA models, and a model checker for properties in the Continuous Stochastic Logic (CSL) — which was introduced in Section 5.2. The tool is freely available under the BSD license, and can be downloaded from `http://www.dcs.ed.ac.uk/pepa/tools/plugin`. Our extension provides the following two views:

1. The *Abstraction View* is a graphical interface that shows the state space of each sequential component in a PEPA model. It provides a facility for labelling states



An outline is not available.

```
// Active badge - 72 states
r = 1.0;
m = 2.0;
s = 1.0;

P14 = (reg14, r).P14 + (move15, m).P15;
P15 = (reg15, r).P15 + (move14, m).P14 + (move16, m).P16;
P16 = (reg16, r).P16 + (move15, m).P15;

S14 = (reg14, infty).(rep14, s).S14;
S15 = (reg15, infty).(rep15, s).S15;
S16 = (reg16, infty).(rep16, s).S16;

DB14 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
DB15 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
DB16 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;

P14 <reg14,reg15,reg16> (S14 <> S15 <> S16) <rep14, rep15, rep16> DB14
```

Concrete States: [4] => Abstract States: [0]

Figure 7.2: The PEPA Plug-In for Eclipse

(so that they can be referred to in CSL properties), and for specifying which states to aggregate.

2. The *Model Checking View* is an interface for constructing, editing, and model checking CSL properties. The property editor provides a simple way to construct CSL formulae, by referencing states that are labelled in the abstraction view. Only valid CSL formulae can be entered.

To illustrate how the tool works, we will use the PEPA model in Figure 7.1 as an example. This is a model of an active badge sensor system, which was first presented in [44], and consists of five sequential components. In the model, a person (component P) moves between three corridors, labelled 14, 15 and 16, which are arranged linearly. Each corridor i has a sensor S_i, which listens for a registration signal reg_i from the person, and informs the database DB. The state of the database effectively records where the person was last seen. The model has three rate parameters — r is the rate at which the badge sends a signal to the sensors, m is the rate of moving between corridors, and s is the rate at which the sensor updates the database.

There are a number of interesting questions we might ask of the model. For example, what proportion of the time does the person spend in each of the corridors? Or, what is the probability that the database can move directly from state DB_{14} to state DB_{16}, missing the fact that the person must have passed through corridor 15? Since this particular example has only 72 states, we can easily analyse it directly without need for abstraction. We will, however, use it as a running example while we describe the features of the tool in Sections 7.1 and 7.2. We will then discuss the architecture of the implementation in Section 7.3, before looking at a slightly larger example in Section 7.4, which better illustrates the power of our abstraction.

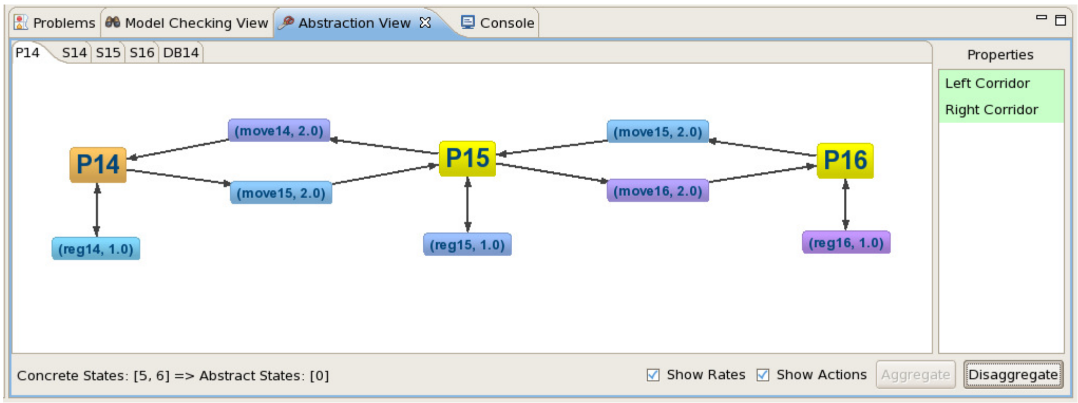
7.1 Specifying State-Based Abstractions

In order to construct and check CSL properties of a PEPA model, we need some way of referring to states of the model. One way of doing this would be to use the names of the sequential component states in the model, but this could lead to very long and cumbersome names — especially if we refer to a large set of states. Ideally, we would prefer to use a single, meaningful name. Our solution is to provide a graphical interface for labelling sets of states.

An overall view of the PEPA plug-in is shown in Figure 7.2. Here, the active badge model of Figure 7.1 is open in the editor, and the abstraction view is in use. The majority of the abstraction view is taken up by a graphical representation of the sequential components in the system equation of the model. In this case, there are five components, and each corresponds to a tab in the view. Currently on display is the database component *DB*.

On the right of the abstraction view is a table showing the atomic properties for the model. Right clicking on this table brings up a menu, from which we can define a new property, or rename or delete an existing one. When we create a new property, the currently selected states in the graph will initially satisfy it, and the other states will not. We can change which properties a state satisfies by right clicking on the state — this allows us to select or deselect the atomic properties, as illustrated in the figure.

An additional feature of the property table is that clicking on a property will highlight all the states that satisfy it. Clicking on a state in the graph will shade the properties that it satisfies in green, and those it does not in red. This allows us to quickly see which states satisfy which properties. It is important to remember that all atomic properties are defined *compositionally*. A state in the system satisfies an atomic property if



Concrete States: [5, 6] => Abstract States: [0]

☑ Show Rates ☑ Show Actions

Figure 7.3: The abstraction interface

and only if its state in each component does. In Figure 7.2, the property "Left Corridor" is satisfied by DB_{14}, but not by DB_{15} and DB_{16}. Since we do not constrain the property for any of the other components, it is satisfied by all states of the system that have the database in state DB_{14}. An example would be the state $P_{14} \parallel S_{14} \parallel S_{15} \parallel S_{16} \parallel DB_{14}$.

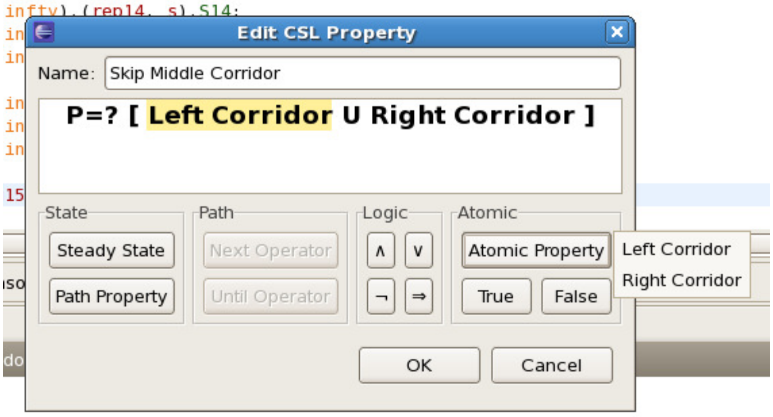
The second function of the abstraction view is, as its name suggests, to specify an *abstraction* — namely, which states to aggregate. Figure 7.3 shows a close-up of the abstraction view, this time for the component P. In this case, we have selected to show both the actions and the rates on the transitions, but since this leads to a more cluttered graph these options are not selected by default.

Aggregating states in a sequential component is simply a matter of selecting the states, and clicking the *Aggregate* button. They can be separated again by clicking *Disaggregate*. Once a set of states have been aggregated, we can only select them as a group — clicking on any one of the states will select them all. Note that the aggregation of states is independent of both the labelling of atomic properties[1] and the definition of CSL properties in the model checking view. This means that we can quickly try out different abstractions — the only thing we need to do is to re-run the model checker each time.

7.2 Model Checking Abstract PEPA Models

To describe properties of a PEPA model, we need a logic for expressing them. To this end, we use the Continuous Stochastic Logic (CSL) [16], which we described in

[1] If we aggregate a set of states when only some of them satisfy a property, the abstract state will have a truth value of '?' (i.e. it 'maybe' satisfies the property).



Edit CSL Property

Name: Skip Middle Corridor

P=? [Left Corridor U Right Corridor]

Figure 7.4: The CSL property editor

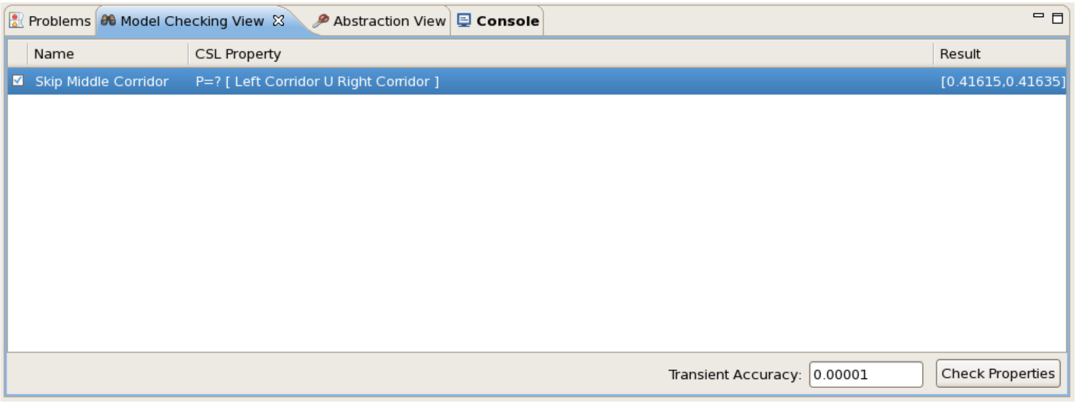
Section 5.2. To illustrate the properties it can express, consider the following formula, for the active badge model from Figure 7.1. Let us assume that we used the abstraction view to define the atomic properties *Left Corridor* and *Right Corridor*, meaning that the database is in states DB_{14} and DB_{16} respectively:

$$\mathcal{P}_{=?}(\mathit{Left\ Corridor}\ \mathcal{U}\ \mathit{Right\ Corridor})$$

This asks the question, "what is the probability that the database will continue to think that the person is in the leftmost corridor, until it becomes aware that the person is in the rightmost corridor?" We can construct this formula using the CSL editor, as illustrated in Figure 7.4.

Classically, for a CTMC, a CSL formula will evaluate to either true or false, or a probability in the case of the test operators Φ_T. In our case, however, we want to model check *abstract* models, hence the test operators will evaluate to a *probability interval*. This describes the best and worst case probability of satisfying the formula, based on the information available in the abstraction. Since this means that we might not know whether or not the model satisfies a property, we need to use a three-valued semantics of CSL [105], with truth values of `tt`, `ff` and ? (true, false, and maybe).

The aim of the CSL editor is to make it as easy as possible to construct a CSL formula. In particular, it ensures that we can construct only well-formed formulae. The buttons on the interface correspond to the various CSL operators and logic connectives, and are enabled by clicking on the part of the formula we want to edit. Hence we cannot enter a path formula where a state formula is required, or vice versa, and the test operators Φ_T can only be used at the top level of a formula. The path and state operator buttons produce a pop-up menu with the available choices — for example,



	Name	CSL Property	Result
☑	Skip Middle Corridor	P=? [Left Corridor U Right Corridor]	[0.41615,0.41635]

Transient Accuracy: 0.00001

Figure 7.5: The model checking interface

timed versus untimed until operators.

The most useful feature of the CSL editor is that it presents us with a list of the atomic formulae that we defined in the abstraction view. Hence, we can easily refer to sets of states in the model, using the labels we created. Because the internal data structures are shared, if we change the name of a property in the abstraction view, it will automatically be updated in the CSL formulae that use it. Similarly, a property cannot be deleted from the abstraction view while it is being used in a formula.

Figure 7.5 shows the model checking view, from which the CSL editor can be opened. The main component of the view is a table of all the properties that are defined for the model. When the *Check Properties* button is pressed, all selected properties are model checked, and the results are displayed next to each property. In this case, we have not abstracted the model, so the result is very precise — a probability interval of $[0.41615, 0.41635]$[2]. The only error here is due to the termination condition of the model checker itself, and can be improved by modifying the *Transient Accuracy* field.

If we require more detailed information about the progress of the model checker, a log is available in the console view. For readability, we reproduce the output of the console, for model checking the above property, in Figure 7.6. In the next section, we will briefly discuss the implementation architecture in more detail, but first let us consider some results obtained by *abstracting* the active badge model — this is, after all, the purpose of our tool.

Table 7.1 shows the result of model checking two transient properties of the active badge model under different abstractions. The first is the same untimed until property we have been considering up to now (where we shorten *Left Corridor* to *Left*, and

[2] We have validated our results for concrete models against PRISM.

```
13:16:04 [badge.pepa] Model added.
13:16:04 [badge.pepa] Model parsed.
13:16:08 [badge.pepa] Kronecker state space derived.  Elapsed time:  5 ms.
13:24:36 [badge.pepa] <Model Checker> Generating abstract CTMC...
13:24:36 [badge.pepa] <Model Checker> Optimising uniformisation constant to 7.0...
13:24:36 [badge.pepa] <Model Checker> Generated abstract CTMC with 72 states.
13:24:36 [badge.pepa] Property "P=?  [ Left Corridor U Right Corridor ]" was checked in 28 ms.
```

Figure 7.6: Console output from the model checker

CSL Property	Aggregated States	State Space Size	Probability Interval
$\mathcal{P}_{=?}(Left\ \mathcal{U}\ Right)$	None	72	[0.41615, 0.41635]
	$\{S_{14}, rep_{14}.S_{14}\}$	36	[0.41615, 0.41635]
	$\{S_{15}, rep_{15}.S_{15}\}$	36	[0.06246, 1.00000]
	$\{S_{16}, rep_{16}.S_{16}\}$	36	[0.00000, 0.90004]
	All of the above	9	[0.00000, 1.00000]
$\mathcal{P}_{=?}(Left\ \mathcal{U}^{[0,1]}\ Right)$	None	72	[0.03018, 0.03019]
	$\{S_{14}, rep_{14}.S_{14}\}$	36	[0.03018, 0.03019]
	$\{S_{15}, rep_{15}.S_{15}\}$	36	[0.01556, 0.03199]
	$\{S_{16}, rep_{16}.S_{16}\}$	36	[0.00000, 0.62023]
	All of the above	9	[0.00000, 0.63213]

Table 7.1: Abstract model checking of the active badge model

Right Corridor to *Right*). The second is a *timed* until property, which states the same condition, but with the additional constraint that the database must enter state DB_{16} (the rightmost corridor) within one time unit. For each property, we investigate the effect of aggregating the states of each of the sensors, and then finally aggregating all three of them at the same time.

The results clearly show how the choice of abstraction affects the precision of the bounds. For both properties, abstracting sensor S_{14} has no effect on the bounds — hence we can halve the size of the model without losing precision. The story for the other sensors is quite different, however. Aggregating S_{16} gives a poor bound in both cases, whereas in the case of S_{15} the bound is much worse for the first property than the second. Finally, aggregating all three sensors results in the largest reduction in the size of the model, but at the cost of limited information for the second property, and no information for the first.

We can intuitively see why aggregating S_{14} has no affect on the precision, since it cannot cause the database to move from its initial state. The other two sensors have the

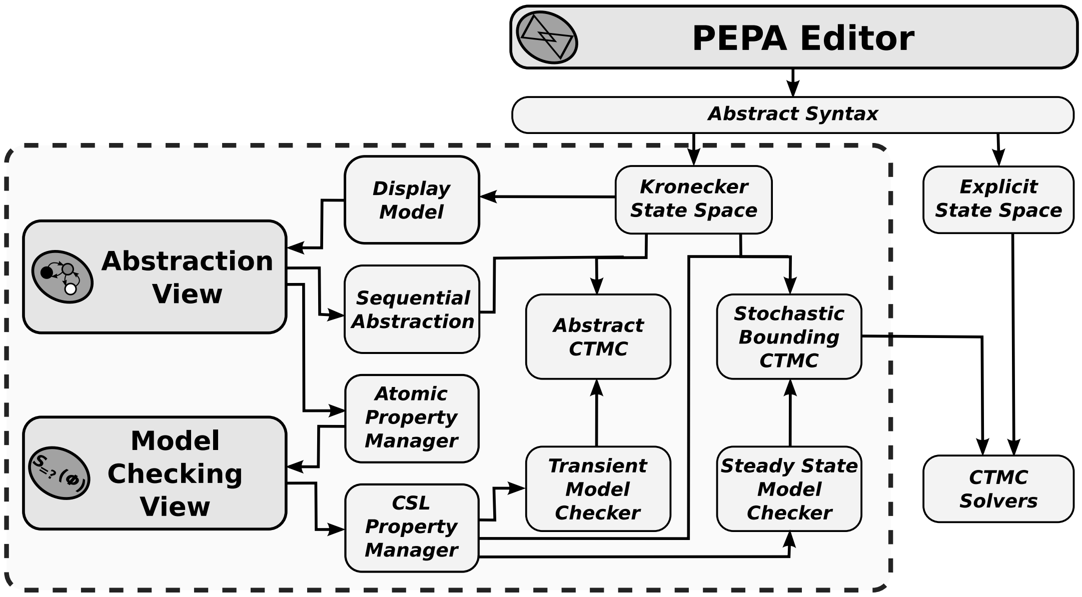


Figure 7.7: The architecture of the abstraction and model checking engine for PEPA

power to move the database to a state that satisfies the property (in the case of S_{16}), or violates the property (S_{15}), hence aggregating either of them will have a big effect on the precision. Having said this, it is not obvious to begin with that it is safe to aggregate S_{14}, and in larger models safe abstractions can be even harder to find.

The advantage of our tool is that it allows us to experiment with different abstractions of the model, without worrying about whether or not it is safe to do so. The results of the model checker are always accurate, in that the actual probability of satisfying the property lies within the interval we obtain. Furthermore, if an abstract model satisfies or violates a particular CSL property, we can be sure that the original model also does. In the worst case, we might obtain imprecise bounds, but if we reduce the size of the model sufficiently, there is very little cost involved.

7.3 Architecture of the PEPA Plug-In

The architecture of our tool follows from a direct implementation of the techniques in Chapter 6, and is shown schematically in Figure 7.7. The dotted box contains our addition to the PEPA plug-in, and we show how it interacts with the parts of the existing tool that we take advantage of — namely, the PEPA editor and parser, and the Markov chain solvers. There are, of course, many other features of the plug-in that are not shown in this diagram.

The key to our approach is in using a *Kronecker* representation of the state space of a PEPA model, which we described in Section 6.1. This forms the main data structure, and all our abstraction techniques take place at the Kronecker level. From this internal representation, we generate a graph of the structure of each sequential component, which we call the *display model*. This is rendered by the abstraction view, which manages a *sequential abstraction* of each component, based on the states that the user is currently aggregating. It also stores the set of atomic properties for the model, which are shared with the model checking view. The model checking view in turn keeps track of a set of CSL properties for each model.

To analyse transient CSL properties, we first construct an abstracted Kronecker state space, and then derive its state space to generate an *abstract CTMC*. Our model checker is explicit state, and the basic algorithm uses a value-iteration approach, as described in [18, 105]. Untimed until properties are verified on the embedded abstract DTMC, since the exit rates do not affect the validity of the property. As explained in Section 5.2, we are unable to model check the timed next operator for abstract Markov chains, due to it not being preserved by uniformisation. We can, however, model check the *untimed* next operator by considering the embedded abstract DTMC before uniformisation.

For steady state CSL properties, the stochastic bounding algorithm operates on the Kronecker state space directly, to compute a lumpable upper bound, from which an aggregated CTMC is generated. To compute the steady state, we use the Matrix Toolkits for Java (MTJ) library [3], which provides a number of solvers including direct solution (Gaussian elimination), the biconjugate gradient method, and the GMRES method. In addition, the PEPA plug-in contains simple implementations of Jacobi and Gauss-Seidel iteration. These solvers are accessed via a factory (in the sense of the design pattern [75]), which makes it easy to substitute one for another — from the user's perspective, this is done by setting an option in the PEPA plug-in.

Because our architecture supports both the abstract Markov chain and stochastic bounding abstractions, it allows us to model check both transient and steady state CSL properties — seamlessly from the point of view of the user.

7.4 A Larger Example

To conclude this chapter, we will examine a larger PEPA model. Figure 7.8 is a model of a round-robin server architecture, where the resources of a single server are shared

$$\begin{array}{lcl}
PC_i & \overset{def}{=} & (arrive, \lambda_i).PC'_i + (walkon_{(i+1)\,\mathrm{mod}\,n}, \top).PC_i \\
PC'_i & \overset{def}{=} & (serve_i, \top).PC_i \\
Server_i & \overset{def}{=} & (walkon_{(i+1)\,\mathrm{mod}\,n}, \omega).Server_{(i+1)\,\mathrm{mod}\,n} + (serve_i, \mu).Server'_i \\
Server'_i & \overset{def}{=} & (walk_{(i+1)\,\mathrm{mod}\,n}, \omega).Server_{(i+1)\,\mathrm{mod}\,n}
\end{array}$$

$$(PC_0 \parallel \ldots \parallel PC_{n-1}) \underset{\{walkon_0,\ldots,walkon_{n-1},serve_0,\ldots,serve_{n-1}\}}{\bowtie} Server_0$$

Figure 7.8: A PEPA model of a round-robin server architecture

between n computers. The server moves around each computer in turn — if there is a job waiting, it services it before moving onto the next computer. Jobs arrive at computer PC_i at rate λ_i, and the service rate of the server is μ. The server moves between computers at rate ω.

Consider this model when $n = 6$, in which case the concrete PEPA model has 768 states. To avoid any symmetry in the model that could allow a more exact aggregation — for example, using compositional bisimulation minimisation [36] — we will assume that every computer has a different arrival rate, $\lambda_i = i + 1$. Table 7.2 shows the results of model checking the following properties:

$$\begin{array}{lcl}
\Phi_1 & = & \mathcal{S}_{=?}(Server') \\
\Phi_2 & = & \mathcal{P}_{=?}(\mathtt{tt}\ \mathcal{U}^{[0,0.1]}\ Server_2)
\end{array}$$

Property Φ_1 looks at the proportion of time spent in a $Server'$ state, where the server has completed a job, but has not yet moved to the next computer. Φ_2 looks at the probability that the server will reach PC_2 within the first 0.1 time units (given that it starts with PC_0).

Looking at the steady state property first, we see that aggregating all the $Server'$ states gives a good upper bound on the actual probability. Although the lower bound provides no information, this would be a useful result if we were interested in verifying that the server spends no more than a certain proportion of time in a $Server'$ state. This is especially true when we consider that the state space has been reduced by 99%. The other choices of aggregation yield poor results, however, which illustrates how the best abstraction depends entirely on the property we are analysing. Note that the extreme reduction in the size of the state space comes about because the steady state abstraction considers only the server component in the worst case context — hence it gives a large reduction in the state space, at the cost of precision.

Property	Aggregated States	State Space	Probability Interval	Verification Time (ms)
Φ_1	None	768	$[0.31184, 0.31184]$	1000
	$\{Server'_{0\ldots5}\}$	7	$[0.00000, 0.33333]$	46
	$\{Server_{0\ldots5}\}$	7	$[0.00000, 0.75000]$	47
	$\{Server'_{2\ldots5}, Server_{3\ldots5}\}$	6	$[0.00000, 1.00000]$	47
Φ_2	None	768	$[0.53940, 0.53941]$	188
	$\{Server'_{0\ldots5}\}$	448	$[0.51954, 0.54567]$	109
	$\{Server_{0\ldots5}\}$	448	$[0.00000, 1.00000]$	109
	$\{Server'_{2\ldots5}, Server_{3\ldots5}\}$	384	$[0.53940, 0.53941]$	110

Table 7.2: Abstract model checking of the round-robin server model

If we look at the second property, by comparison, we see that we achieve the best results when we abstract all the states following $Server_2$, but before $Server_0$. In this case, we can halve the state space without affecting the precision. In fact, since Table 7.2 only looks at aggregating states on the server, we can do even better. If we aggregate the computer states for $PC_2 \ldots PC_5$, we can reduce the state space to just 24 states (a 97% reduction in size) without affecting the precision. The reason we can achieve such good results here, is that after the server passes through the $Server_2$ state, the property must be satisfied — hence we can ignore all subsequent states. The abstraction view allows us to take advantage of this aggregation very quickly, without requiring any modifications to the model.

The main limitation with our tool at present is that we have not optimised the data structures for the explicit represention of the state space of a model. We only use an in-memory representation (as opposed to, for example, using disk storage), and we find that we run out of memory when we deal with models that have tens of thousands of states. Our abstraction techniques can scale easily to very large models, since they operate on the Kronecker representation of the model, but we need to improve the efficiency of the model checker — particularly with regard to memory usage — if our tool is to be of practical use. The main reason for developing this tool, however, was to demonstrate the applicability of our abstraction, and to that end we have succeeded. There is clearly much future work to be done, however, in improving its functionality and performance.

Chapter 8

Conclusions

"*Software developers care too little about performance until it is too late!*"

Given the motivation of this thesis, one might be forgiven for jumping to such a provocative conclusion. But although performance evaluation is often under-prioritised in software development projects, to say that the developers do not *care* about performance is not only wrong, but completely misses the point. The fact is that they simply do not have access to many of the tools, methods, and techniques that would enable them to reason about performance. Hence we could just as easily jump to an equally provocative conclusion, by saying that computer scientists and performance modellers care too little about supporting real-world software development!

Of course, this statement is just as blatantly untrue as the first. In reality, both software developers and performance modellers care very strongly about bridging the gap between theory and practice. In the case of developers, there is a clear need for something more disciplined than current performance testing and tuning practices, even if they do not know where to look for a solution. In the case of performance modelling, there have been concerted efforts to promote disciplines such as software performance engineering, and to include performance information in specification languages such as UML. We spent the majority of Chapter 2 talking about such issues, and it should be clear that people have been working on these problems for a considerable time.

The goal of this thesis, however, has been to motivate the next step in bridging this gap. If developers are to fully utilise the capabilities of performance modelling and evaluation, they need techniques that are as easy to use as possible, and that relate to the languages that they are already used to. This ultimately means that they need *tool support* that can deal directly with *program code*, if the uptake is to be as great as possible. Whilst we are a long way from a practical solution to this problem, our current knowledge in fields such as program analysis, performance modelling, and stochastic model checking is at a stage where we should be making progress in addressing it.

At the outset in Chapter 1, we proposed a framework for *performance-driven development*, where the central idea is to abstract program code to a performance model. By looking at a hypothetical software engineering process that could be used in conjunction with more traditional qualitative methods, we could then try to address some of the challenges involved. To this end, our focus in this thesis has been on two of the most important problems: firstly, how to automatically extract a performance model from program code, and secondly, how to reduce the size of large performance models so that they can be analysed. Central to both is the notion of *abstraction*.

Of course, these two problems are too general to be tackled directly, and so we have restricted our attention to one specific programming language and one specific modelling language. Our programming language — Siril, the Simple Imperative Remote Invocation Language — allows statically defined objects, or resources, which each consist of a number of methods that can be invoked via remote procedure calls. Methods have local integer variables, and can perform linear arithmetic operations on them, but we do not allow looping behaviour at this point. Our modelling language — PEPA, the Performance Evaluation Process Algebra — is a simple and widely used formalism for compositionally specifying continuous time Markov chains.

The contributions of this thesis can be split into two parts, corresponding to these two problems. Chapters 3 and 4 showed how we can use *abstract interpretation* in combination with *measure theory* to derive a PEPA model from a Siril program. Chapters 5, 6 and 7 then showed how to *compositionally abstract* PEPA models with a view to *model checking* properties in the Continuous Stochastic Logic (CSL). One of the interesting ideas here is that we effectively combine the strengths of static analysis and model checking — static analysis has the advantage of being able to work directly with program code, whereas model checking has the advantage of using temporal logics to allow a rich specification of properties. In our case, we use static analysis to derive a more abstract representation of the program, and model checking to verify it with respect to some property of interest.

In this chapter, we will begin by summarising the main results of the thesis in Section 8.1. In Section 8.2 we will discuss in more detail how our techniques could be built upon and extended in the future, in addition to looking at the major challenges of performance-driven development that are yet to be faced. Finally, we will end with some concluding remarks, in Section 8.3.

8.1 Summary of Results

The main results and ideas presented in this thesis are as follows. We began in Chapter 1 by motivating the need for performance-driven development, and outlining the structure of a software development process through which this could take place. This consists of four main stages — abstraction from code to a performance model, specialisation of the model, analysis of the model, and refinement of the code based on the results of the analysis.

In Chapter 3 we introduced the Siril language, and a semantics in terms of probabilistic automata. This is a novel extension of Kozen's semantics of probabilistic programs, where a program is viewed as a continuous linear operator over measures — in general, mapping a distribution over the initial state of a program's variables to a subprobability measure corresponding to the state of the program if it terminates. While Kozen's semantics is entirely denotational, our semantics introduced an automaton structure, where transitions between stages are labelled with measure transformers. This can be viewed in two ways — either as a purely computational device for describing operators over measures, or as a control flow graph structure, where stages correspond to program points.

After developing this semantics, we proved that the outgoing transitions of a stage preserve the total measure. This allowed us to present a discrete time probabilistic interpretation of the semantics, in which transitions are labelled with a probability — given by the ratio of the total measure before and after the transition. A continuous time interpretation can similarly be constructed by assigning rates to stages in the probabilistic automaton. We then discussed how we might construct a collecting semantics that projects the probabilistic interpretation onto the state space of an individual method, so as to regain the compositionality of the Siril program.

Since it is computationally infeasible to represent and operate on arbitrary measures, we presented an abstract interpretation of Siril programs in Chapter 4. The key idea is to use truncated multivariate normal measures, which can be compactly represented, and are closed under abstract linear and truncation operators. Intuitively, the underlying multivariate normal measure records the dependencies between variables, and the truncation intervals record constraints on the values of the variables. We presented an abstract semantics of Siril, and an abstract interpretation that induces a safe over-approximation of the concrete interpretation, with respect to the measures that occur at each state.

To construct a PEPA model from our abstract interpretation, we presented an abstract collecting semantics. This takes place in two stages — we first project the abstract interpretation onto the state space of each individual method (labelling the transitions with input and output measures), then we map each projected transition system onto a sequential PEPA component. In constructing an overall PEPA model from these components, we showed how a number of simple transformations at the modelling level can be used to analyse the program's performance in the context of multiple users and resources. Finally, we discussed how our approach could be modified to generate an MDP-based model containing non-determinism.

In Chapter 6, we presented a compositional abstraction framework for model checking CSL properties of PEPA models. We began by presenting a general Kronecker form for PEPA, that allows the generator matrix of the underlying CTMC to be described compositionally. Our abstraction combines two different techniques — abstract Markov chains, which can be used to analyse transient CSL properties, and stochastic bounds, which can be used for steady state properties. In both cases, we showed how to compositionally construct an abstraction of a PEPA model, and proved that the abstraction is safe. In the case of stochastic bounds, we generalised the algorithm of Fourneau *et al.* so that it can be applied to a partially ordered state space, and to the particular stochastic ordering constraints that our abstraction requires.

We have implemented a tool for abstracting and model checking PEPA models — described in Chapter 7 — which is an extension to the PEPA plug-in for Eclipse. This provides a graphical interface for specifying sets of states to aggregate in a componentwise fashion, and for labelling atomic properties so as to make it easier to construct CSL formulae.

8.2 Evaluation and Future Work

In this section, we will examine some of the ways in which the work in this thesis can be extended and improved upon, as well as identifying some directions for future research. In Sections 8.2.1 and 8.2.2 we will look at the two main areas of contribution in this thesis — respectively, stochastic abstraction of programs, and stochastic abstraction of performance models. We will then briefly discuss the relationship between static analysis and model checking in Section 8.2.3, before looking at how we could build tool support to facilitate performance-driven development in Section 8.2.4.

8.2.1 Stochastic Abstraction of Programs

We have seen in Chapters 3 and 4 of this thesis how to extract a PEPA model from a language called SIRIL. This is quite a simple language, in that it allows only immutable objects, remote procedure call (RPC) style invocation, integer variables, and loop-free method bodies. Despite this, we can perform some useful analyses of distributed systems whose inputs are governed by a probability distribution. There are many interesting and open research problems that need to be addressed, however, before we can apply our techniques to industrially relevant programming languages.

Recall that when we presented our framework for performance-driven development in Chapter 1, we identified SIRIL as an *intermediate language* — between an implementation language and a performance modelling formalism. With this in mind, we will discuss two key areas for future work in this section. The first is how we might extend the SIRIL language and its analysis so as to relax the restrictions in this thesis. The second is how we can move to real implementation languages such as C, C++, and Java, which involves dealing with more complex language features. We will consider each of these in turn, starting with the extensions to SIRIL.

From a practical point of view, the most important features that SIRIL currently lacks are *loops* and *mutable objects*. Note that if we had both of these features, we could easily model a more general message-passing style of communication by using mutable 'channel' objects. Sending a message would correspond to calling a method that changes the object's state, and receiving a message would correspond to a method that queries its state — either returning immediately with the possibility of failure (an asynchronous receive), or blocking until there is a message to receive, by polling the state (a synchronous receive). Furthermore, we can always program non-linear arithmetic operations if we have loops in our language. We previously attempted to analyse both loops and mutable objects using the techniques in this thesis, but we ran into a number of difficulties, which we will now discuss.

When we first presented the ideas of Chapter 4 in [170], we did so in terms of a language containing `while`-loops. As far as the concrete and abstract semantics of this language is concerned, there is little difference to that presented in this thesis — we just include additional states in the probabilistic automaton of a program, to account for re-entering loops. We describe this in Appendix A. For the abstract interpretation, however, the situation is much more complicated. We cannot simply 'run' the abstract interpretation as it stands, since there is no guarantee that it will terminate in the presence of loops. To ensure termination, we need a way of over-approximating the possible measures within a loop, using a *widening operator* [157].

This is a standard device in abstract interpretation, but the nature of our domain makes it difficult to apply generally. If we want to avoid expanding out a loop, we could consider associating a *multiset* of measures with each stage in the probabilistic automaton of a method. Intuitively, this means that we store the measures for all iterations of the loop. For simple classes of loop, such as an iterative counter, where we can predict what the future measures will be given the current measure, suitable widening functions allow us to 'jump' to a set of measures that over-approximates these [170]. Unfortunately, this approach has a number of drawbacks:

1. We need a compact way of representing multisets of measures.

2. We require a widening operator that is individually tailored for each loop, if we are to avoid too gross an over-approximation. In general, this requires us to infer *loop invariants*, and this is made particularly difficult when there are conditional statements within the loop.

3. When we exit a loop, we would like to return to a single measure, so that it can be transformed by future measure operators. Unfortunately, such a measure will be *mixed* truncated multivariate normal in general, which does not exhibit the same nice properties as a truncated multivariate normal measure.

It is interesting to add to the latter point that mixed multivariate normal distributions are routinely used in the context of Hidden Markov Models (HMMs) with Gaussian observations [148]. It might therefore be possible to use some of the techniques from this area in order to deal with mixed measures.

Based on the difficulties that we encountered, it appears that a direct extension of our technique in the context of loops is not a good way to proceed. That is not to say that all is lost, however — it just means that we need to look at different techniques for this situation. One idea might be to first transform loops so that the operation of the body is idempotent, in that executing it multiple times gives the same result as executing it once. This would mean that we could ignore the possibility of re-entering the loop when computing the measures. Such a transformation would clearly not preserve the semantics of the program, however, and so we would need to make some appropriate abstraction — such as taking the 'average' values of loop-counters, and making the guard of the loop a pure probabilistic choice. The key idea is to combine a different static analysis technique with our abstraction, and it would be interesting to investigate the extent to which this is possible.

Compared to the problem of loops, analysing programs with mutable objects appears to be a much more straightforward extension to SIRIL. Indeed, in the absence of loops, we could apply the same approach of Chapter 4 — performing a system-level abstract interpretation, and regaining compositionality through the collecting semantics. This would just require some minor modifications to maintain the state of each object in the corresponding PEPA component. The real difficulty, however, is when we want to combine mutable objects and loops, for which we need a better solution.

An approach that we considered in the past was to try to perform the abstract interpretation itself *compositionally*. In other words, rather than interpreting the program as a whole, we can interpret each method individually to reach a local fixed point, then compute a global fixed point of the system by chaotic iteration [51]. Such an approach, however, introduces a great many additional problems — for example, we need to keep track of temporal dependencies between variables if we want to avoid considering far more execution paths than are possible in practice. This is essentially the same problem as found in interprocedural data-flow analyses [133], except that we additionally have a more complex domain of truncated multivariate normal measures.

An additional limitation of the approach presented in this thesis is that we store and manipulate measures over the *entire state space* of the system. While this simplifies our mathematical presentation, it is not very practical, as it means that we always record the state of every variable in every method, leading to very large data structures for each measure. In most programs, however, there is limited coupling between variables in different methods, and so we should hope to find much more efficient representations. It makes sense, however, to only do so at the implementation level, since the explicit representation is mathematically more straightforward to reason about.

We have now discussed in some detail how the SIRIL language and our analysis could be extended to a richer set of language features. This is still a long way from real implementation languages, however, and so we need to find ways to bridge this gap. To this end, we will consider how we might develop abstractions from languages such as C, C++, and Java into SIRIL, or an extension of it.

The difficulty with many program languages is that they lack mathematical elegance in favour of practical features. This is not so much of a problem for Java, but C, for instance, does not enforce type safety, and its semantics is in some places ambiguous — to the extent that different compilers produce different results. Fortunately, there has been a great deal of work in attempting to address such problems — in particular, the C Intermediate Language (CIL) [130] was developed as a means of transforming programs into a simplified subset of C that avoids these problems. In light of such tools, the main research challenges are to deal with more general language features such as enriched sets of primitive types, data structures on the heap, and object orientation.

In relation to the first of these points, it would be quite straightforward to include additional primitive types such as Booleans and floating points in SIRIL. If we take the abstraction that a floating point is a real number — i.e. we ignore numerical issues — then our semantics can deal with this case already[1]. Boolean variables could be modelled as discrete random variables, by viewing them as predicates that hold with a certain probability. This would require some modifications to our domain of measures, but would otherwise be a straightforward extension of SIRIL.

The main challenge for future research is to deal with the heap — such as pointers, more complex data structures, and virtual methods. For these, there already exist various static analysis techniques, such as the Three-Valued Logic Analyser (TVLA) [117] and separation logic [149]. These have been applied to applications such as shape analysis of linked lists [61] and trees [39], and analysis of pointer arithmetic [38]. We would therefore need to look at ways of combining these techniques with ours — ideally, we would like to know the *probability* that a pointer points to a particular location. Note that handling object orientation essentially amounts to being able to deal with mutable objects, more complex data structures, and pointers (in the sense of virtual methods and inheritance).

Finally, an important practical issue is to determine which parts of a program to analyse — from the point of view of constructing a performance model — and what to do about partially-completed code. In [169], we suggested a system of annotations, allowing the developer to specify such information directly. This is not an ideal approach, however, since it potentially requires a lot of work on the part of the developer. We will discuss some possible approaches to this problem in more detail in Section 8.2.4, when we talk about tool support for performance-driven development.

8.2.2 Stochastic Abstraction of Performance Models

There are many techniques for abstracting performance models, but in this thesis we have focussed on just two approaches in particular — abstract Markov chains, and

[1] We would only need to modify the semantics of comparison in this case, by eliminating the conversion of an integer constant c to a real interval $[c-0.5, c+0.5]$.

stochastic bounds — and how they can be applied compositionally to PEPA models. This idea of compositional abstraction allows us to avoid constructing the entire state space of the model, and can therefore allow us to analyse much larger systems than before. This idea could in theory be applied to any compositional performance modelling formalism, and not just PEPA — for example, abstract Markov chains have recently been applied compositionally to Interactive Markov Chain (IMC) models [104].

The main difficulty with state-based abstraction techniques, however, is that their precision depends very much on *which* states we try to aggregate. Our approach was to develop a graphical interface to allow the modeller to quickly experiment with different aggregations, which we presented in Chapter 7. Ultimately, however, we would like to be able to *automatically* select a good choice of states to aggregate.

One recent approach that has been successful is to extend the ideas of counterexample-guided abstraction-refinment (CEGAR) [22, 86] to probabilistic models [89, 107, 182]. As in the qualitative setting, the idea is to begin with an initial coarse abstraction, which is then incrementally refined by eliminating spurious counterexamples. Eventually, either the property of interest will be verified, or a real counterexample will be found. Unlike in the qualitative setting, however, a probabilistic counterexample is a *set* of paths whose combined probability violates a property, rather than an individual path. There has been some work on compact representations for these counter-examples, using regular expressions [55].

This approach, however, is restricted to *probabilistic reachability* properties, as opposed to performance properties in general. These are certainly useful though, as they include the transient properties of CSL. It would be of interest to see whether we could combine this approach with our compositional abstraction, so that it could potentially be applied to larger models.

When verifying *steady state* properties, abstraction techniques are much more difficult in general, as can be seen by the number of constraints we needed to impose in Chapter 6, so as to compositionally construct stochastic bounds. Due to the quite strict requirements of the strong stochastic ordering, there has been some work on weakening this, particularly in the context of Markov reward models [53, 52]. The idea is to relax the requirement of montonicity by replacing it with a weaker condition — if two states are related under a partial ordering on the state space, then the greater state must “cover” the lesser state with respect to the accumulated reward. This is a weaker condition than monotonicity over a partial ordering, as in this thesis, and it would be worthwhile to see whether our compositional approach for PEPA could be lifted to it.

In practice, most applications of stochastic bounds are in relation to particular models for which specialised bounds can be constructed for specific properties [37, 119]. We have found with our approach that general algorithms tend to give very loose bounds much of the time, which would suggest that stochastic bounds are not a complete solution to our problem. They are, however, useful for quickly obtaining rough bounds in relation to individual components of the model. One interesting direction for future research would be to look at steady state properties in terms of *long-run averages* of an MDP [7]. This would allow us to verify such properties by adapting the existing algorithms for model checking MDPs and abstract Markov chains — and potentially, also the probabilistic CEGAR approaches.

In this thesis we have concentrated on *safe abstractions*, but an alternative approach we might take is to look for *close abstractions*. This is characterised by the work on probabilistic abstract interpretation [141], which applies the idea of abstract interpretation to a vector space rather than a complete lattice. This gives a natural notion of metrics, allowing the concept of a "closest" abstraction. It would be valuable to investigate whether such techniques can be applied in our setting — one question in particular is whether steady state properties can be compared under this approach.

8.2.3 Static Analysis versus Model Checking

In this thesis, we have looked at both static analysis and model checking in the context of performance. As we commented in Chapter 4, there have been relatively few applications of static analysis in the context of probabilistic and stochastic systems, in contrast to the vast literature on probabilistic and stochastic model checking. The main reason for this is most likely that *performance* is a property of an entire system, including both the program code and the environment that it runs in. Hence it is more natural to construct a *model* of the system, which includes the environment, and to verify performance properties using model checking.

Even given this dependency on the environment, however, it is possible to analyse *probabilistic* properties of a program using a combination of static analysis and model checking. One example of this is probabilistic CEGAR, which we discussed in the previous section, and another is the static analysis of Chapter 4, coupled with stochastic model checking, as presented in this thesis. In both cases, static analysis is used as a means of constructing a performance model, which is then verified in relation to a particular property. The main difference, aside from the abstractions used, is that probabilistic CEGAR has a feedback loop — the model is generated with a fixed property in mind, and then refined in relation to this property, using counterexamples from model checking. Our approach gives a more general model, for which we can model check a wider range of properties, at the cost that the model may be much larger.

In the context of quantitative analysis of software, it seems likely that many of the major advances in the future will come from an application of *both* static analysis and model checking. This is therefore an important research area to pursue.

8.2.4 Tool Support for Performance-Driven Development

As we have repeated on a number of occasions throughout this thesis, tool support is essential if we are to make performance-driven development viable in practice. In particular, it is important to support the integrated development environments that are popular with developers, such as Visual Studio [4] and Eclipse [1]. In Chapter 7, we described an Eclipse plug-in for abstracting and model checking PEPA models, but to really make our techniques accessible to developers, we need direct tool support at the level of program code.

Although we did not present an implementation of the techniques in Chapters 3 and 4 in this thesis, we did develop the theory to support it. We discussed the challenges of lifting these techniques to industrial implementation languages in Section 8.2.1, but we additionally need an intuitive and easy-to-use *interface* for developers. There are two key aspects here — gathering the required information to carry out a performance analysis, and presenting the results of the analysis back to the developer.

The first of these entails asking the developer for information — most importantly, which parts of the code to analyse, how to interpret partially-written code, and what properties to check for. The first two could be specified using annotations, but it might be difficult to convince developers to invest the time and effort to do so. If there exist specification documents such as UML diagrams, we could instead try to extract the information — for example, only modelling the classes used in a particular diagram, or using timing information from the specification when there is no code available.

Specifying performance properties is also very difficult to make intuitive. The approach that we took in Chapter 7 can be useful for modellers, but logics such as CSL are certainly not intuitive to developers. Similarly, approaches such as performance trees [175] and probabilistic specification patterns [83] still require a certain understanding of probabilities and statistics that developers may not have. One idea that might be worth pursuing is to construct an analogue of the unit testing framework. In other words the developer writes a *test* that invokes the system in a certain way, and either collects some performance information (such as the utilisation of a server) or asserts that certain performance requirements must hold (such as the server responding within one second with a 99.9% probability). This could be based on the ideas in [6], for specifying the probability of observing events, and the time between them. Finding an appropriate language to express this would of course require careful consideration.

Once we have verified a performance property of the program, we then need to relate this information back to the developer. The result itself is easy to convey — for example, pass or fail — but the challenge is to explain *why* that is the case. There is a great deal of scope for adapting interactive debuggers to present such information, but this is really a very open problem, and it is not clear what the best approach would be.

Overall, there are a great many questions to be answered in developing tool support for performance-driven development, some that are research problems, and others that are engineering challenges. But if we can develop performance analysis techniques to the extent that they can be applied to industrial development projects, the time invested in answering these problems will more than pay for itself.

8.3 Concluding Remarks

We began this thesis with a grand vision — one in which software developers have the tools and techniques at their disposal to be able to build distributed software systems that are *engineered* to perform. Our goal was to demonstrate that current techniques in static analysis, performance modelling, model checking, and abstraction can be combined and used to make this vision a reality. In doing so, we have developed various new ideas — such as using truncated multivariate normal distributions in program analysis — and extended various existing ideas — such as making certain Markov chain abstractions compositional. But the most important contribution is that we have illuminated the potential to bring our techniques to developers. It is simply unrealistic to expect the developers to come to us.

In this thesis we have made an important step towards our vision of performance-driven development, despite there being many research problems that remain open. Yet this step could be the first of many, and we hope that by encouraging the combined application of a wide variety of techniques, the computer science community as a whole will be able to revolutionise the treatment of performance in software development.

Bibliography

[1] The Eclipse platform. `http://www.eclipse.org`.

[2] JUnitPerf. `http://clarkware.com/software/JUnitPerf.html`.

[3] Matrix Toolkits for Java (MTJ). `http://code.google.com/p/matrix-toolkits-java/`.

[4] Microsoft Visual Studio 2008. `http://www.microsoft.com/visualstudio`.

[5] O. Abu-Amsha and J.-M. Vincent. An algorithm to bound functionals of Markov chains with large state spaces. In *4th INFORMS Conference on Telecommunications*, 1998.

[6] L. De Alfaro. Temporal logics for the specification of performance and reliability. In *STACS '97: Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1200 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 1997.

[7] L. De Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford, CA, USA, 1998.

[8] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[9] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[10] S.W. Ambler. *The Object Primer: Agile Model Driven Development with UML 2*. Cambridge University Press, 2004.

[11] H.H. Ammar and S.M. Rezaul Islam. Time scale decomposition of a class of generalized stochastic Petri net models. *IEEE Transactions on Software Engineering*, 15(6):809–820, 1989.

[12] A.W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1997.

[13] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 269–276. Springer-Verlag, 1996.

[14] D.A. Bacigalupo, S.A. Jarvis, L. He, D.P. Spooner, D. Pelych, and G.R. Nudd. A comparative evaluation of two techniques for predicting the performance of dynamic enterprise systems. In *Parallel Computing: Current & Future Issues of High-End Computing, (Proceedings of the International Conference ParCo 2005)*, pages 163–170. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[15] C. Baier. On algorithmic verification methods for probabilistic systems. Habilitation Thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[16] C. Baier, B.R. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 358–372, Chicago, IL, USA, 2000. Springer-Verlag.

[17] C. Baier, B.R. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[18] C. Baier, H. Hermanns, J.-P. Katoen, and B.R. Haverkort. Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. *Theoretical Computer Science*, 345(1):2–26, 2005.

[19] C. Baier, J.-P. Katoen, H. Hermanns, and V. Wolf. Comparative branching-time semantics for Markov chains. *Information and Computation*, 200(2):149–214, 2005.

[20] G. Balbo. Introduction to stochastic Petri nets. In *Lectures on Formal Methods and Performance Analysis*, pages 84–155, Berg en Dal, The Netherlands, 2002. Springer-Verlag.

[21] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[22] T. Ball and S.K. Rajamani. The SLAM toolkit. In *Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[23] P. Ballarini. *Towards Compositional CSL Model Checking*. PhD thesis, Dipartimento di Informatica, Università di Torino, 2004.

[24] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, 1975.

[25] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[26] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[27] A. Benoit, B. Plateau, and W.J. Stewart. Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems. *Future Generation Computer Systems*, 22(7):838–847, 2006.

[28] M. Bernardo and R. 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.

[29] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003.

[30] B.W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[31] H. Bohnenkamp, P.R. D'Argenio, H. Hermanns, and J.-P. Katoen. MoDeST: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.

[32] R.J. Boucherie. A characterization of independence for competing Markov chains with applications to stochastic Petri nets. *IEEE Transactions on Software Engineering*, 20(7):536–544, 1994.

[33] J. Bradley and S. Gilmore. Stochastic simulation methods applied to a secure electronic voting model. In *PASM '05: Proceedings of the 2nd Workshop on Practical Applications of Stochastic Modelling*, pages 127–149, Jul 2005.

[34] J. T. Bradley and W.J. Knottenbelt. The ipc/HYDRA tool chain for the analysis of PEPA models. In *QEST '04: Proceedings of the Quantitative Evaluation of Systems, First International Conference*, pages 334–335, Enschede, The Netherlands, 2004. IEEE Computer Society Press.

[35] P. Buchholz. Markovian Process Algebra: Composition and equivalence. In *Proceedings of the 2nd Workshop on Process Algebra and Performance Modelling (PAPM '94)*, pages 11–30, Erlangen, 1994.

[36] P. Buchholz and P. Kemper. Efficient computation and representation of large reachability sets for composed automata. *Discrete Event Dynamic Systems*, 12(3):265–286, 2002.

[37] A. Busic, M.B. Mamoun, and J.-M. Fourneau. Modeling fiber delay loops in an all optical switch. In *QEST '06: Proceedings of the Third International Conference on the Quantitative Evaluation of Systems*, pages 93–102, Riverside, California, USA, 2006. IEEE Computer Society Press.

[38] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Static Analysis Symposium (SAS)*, pages 182–203. Springer-Verlag, 2006.

[39] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 271–282. ACM, 2005.

[40] J. Campos, J.M. Colom, H. Jungnitz, and M. Silva. Approximate throughput computation of stochastic marked graphs. *IEEE Transactions on Software Engineering*, 20(7):526–535, 1994.

[41] C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance modelling with the Unified Modelling Language and stochastic process algebras. *IEE Proceedings: Computers and Digital Techniques*, 150(2):107–120, Mar 2003.

[42] M. Capiński and E. Kopp. *Measure, Integral and Probability*. Springer-Verlag, second edition, 2004.

[43] R. Chadha, M. Viswanathan, and R. Viswanathan. Least upper bounds for probability measures and their applications to abstractions. In *CONCUR '08: Proceedings of the 19th International Conference on Concurrency Theory*, volume 5201 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[44] G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. In *ARTS '99: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, pages 211–227. Springer-Verlag, 1999.

[45] G. Clark and J. Hillston. Product form solution for an insensitive stochastic process algebra structure. *Performance Evaluation*, 50(2–3):129–151, Nov 2002.

[46] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer-Verlag, 2004.

[47] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from UML diagrams. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 58–70, Ottawa, Ontario, Canada, 2000. ACM.

[48] T. Courtney, D. Daly, S. Derisavi, V. Lam, and W.H. Sanders. The Möbius modeling environment. In *Tools of the 2003 Illinois International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems*, pages 34–37, 2003.

[49] P.J. Courtois. *Decomposability: Queueing and Computer System Applications*. Academic Press, 1977.

[50] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[51] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 1–12, Rochester, NY, USA, 1977. ACM.

[52] D. Daly. *Bounded aggregation techniques to solve large Markov models*. PhD thesis, Champaign, IL, USA, 2005.

[53] D. Daly, P. Buchholz, and W.H. Sanders. Bound-preserving composition for Markov reward models. In *QEST '06: Proceedings of the Third International Conference on the Quantitative Evaluation of Systems*, pages 243–252, Riverside, California, USA, Sep 2006. IEEE Computer Society Press.

[54] A. D'Ambrogio and P. Bocciarelli. A model-driven approach to describe and predict the performance of composite services. In *WOSP '07: Proceedings of the 6th International Workshop on Software and Performance*, pages 78–89, Buenes Aires, Argentina, 2007. ACM.

[55] B. Damman, T. Han, and J.-P. Katoen. Regular expressions for PCTL counterexamples. In *QEST '08: Proceedings of the Fifth International Conference on Quantitative Evaluation of Systems*, pages 179–188, St. Malo, France, 2008. IEEE Computer Society.

[56] P.R. D'Argenio, B. Jeannet, H.E. Jensen, and K.G. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *Proceedings of the Joint International Workshop, PAPM-PROBMIV 2001*, volume 2165 of *Lecture Notes in Computer Science*, pages 39–56. Springer-Verlag, 2001.

[57] T. Dayar and W.J. Stewart. Quasi-lumpability, lower-bounding coupling matrices, and nearly completely decomposable Markov chains. *SIAM Journal on Matrix Analysis and Applications*, 18(2):482–498, 1997.

[58] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. *SIGSOFT Software Engineering Notes*, 29(1):94–103, 2004.

[59] P.J. Denning and J.P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, 1978.

[60] E. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, Oct 1972.

[61] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? on the automated verification of linked list structures. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 250–262. Springer-Verlag, 2004.

[62] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computing Programming*, 2:241–266, 1982.

[63] H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In *Proceedings of SPIN'06*, volume 3925 of *Lecture Notes in Computer Science*, pages 71–88, 2006.

[64] P. Fernandes, B. Plateau, and W.J. Stewart. Efficient descriptor-vector multiplications in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.

[65] V. Firus, S. Becker, and J. Happe. Parametric performance contracts for QML-specified software components. *Electronic Notes in Theoretical Computer Science*, 141(3):73–90, 2005. Proceedings of Formal Foundations of Embedded Software and Component-Based Software Architectures.

[66] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nielson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of Programming Language Design and Implementation (PLDI) 2002*, pages 234–245, Berlin, Germany, Jun 2002. ACM.

[67] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, second edition, 2004.

[68] J.-M. Fourneau and L. Kloul. A precedence PEPA model for performance and reliability analysis. In A. Horváth and M. Telek, editors, *Formal Methods and Stochastic Models for Performance Evaluation: Third European Performance Engineering Workshop (EPEW 2006)*, volume 4054 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Jun 2006.

[69] J.-M. Fourneau, M. Lecoz, and F. Quessette. Algorithms for an irreducible and lumpable strong stochastic bound. *Linear Algebra and its Applications*, 386:167–185, 2004.

[70] J.-M. Fourneau, B. Plateau, and W.J. Stewart. Product form for stochastic automata networks. In *ValueTools '07: Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools*, pages 1–10, Nantes, France, 2007. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST).

[71] G. Fox. Performance engineering as a part of the development life cycle for large-scale software systems. In *ICSE '89: Proceedings of the 11th International Conference on Software Engineering*, pages 85–94, Pittsburgh, Pennsylvania, United States, 1989. ACM.

[72] G. Franceschinis and R.R. Muntz. Bounds for quasi-lumpable Markov chains. *Performance Evaluation*, 20(1-3):223–243, 1994.

[73] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009.

[74] S. Frolund and J. Koistinen. QML: A language for quality of service specification. Technical Report HPL-98-10, Hewlett Packard Software Technology Laboratory, 1998.

[75] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[76] A. Genz. Numerical computation of multivariate normal probabilities. *Journal of Computational and Graphical Statistics*, 1(2):141–149, 1992.

[77] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[78] S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794 of *Lecture Notes in Computer Science*, pages 353–368, Vienna, Austria, May 1994. Springer-Verlag.

[79] N. Götz, U. Herzog, and M. Rettelbach. TIPP — introduction and application to protocol performance analysis. In H. König, editor, *Formale Beschreibungstechniken für verteilte Systeme*, FOKUS Series, pages 105–125. Saur Publishers, 1993.

[80] R.M. Graham, G.J. Clancy Jr., and D.B. DeVaney. A software design and evaluation system. *Communications of the ACM*, 16(2):110–116, 1973.

[81] Object Management Group. UML profile for schedulability, performance, and time specification. *OMG Specification*, Jan 2005.

[82] Object Management Group. UML profile for modeling and analysis of real-time and embedded systems (MARTE). *OMG Specification*, Jun 2008.

[83] L. Grunske. Specification patterns for probabilistic quality properties. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 31–40, Leipzig, Germany, 2008. ACM.

[84] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. *Proceedings of the Real Time Systems Symposium*, pages 102–111, Dec 1989.

[85] A. Henriksson and H. Larsson. A definition of round-trip engineering. Technical report, Linköping University, Sweden, 2003.

[86] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[87] H. Hermanns. *Interactive Markov Chains*. PhD thesis, Universität Erlangen-Nürnberg, 1998.

[88] H. Hermanns, D. Jansen, and Y. Usenko. From StoCharts to MoDeST: A comparative reliability analysis of train radio communications. In *Proceedings of the 5th International Workshop on Software and Performance*, pages 13–23, Palma, Spain, 2005. ACM.

[89] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In *CAV '08: Proceedings of the 20th International Conference on Computer Aided Verification*, pages 162–175, Princeton, NJ, USA, 2008. Springer-Verlag.

[90] J. Hillston. The nature of synchronisation. In *Proceedings of the 2nd Workshop on Process Algebra and Performance Modelling (PAPM '94)*, pages 51–70, Erlangen, 1994.

[91] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[92] J. Hillston. Fluid flow approximation of PEPA models. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 33–43, Torino, Italy, Sep 2005. IEEE Computer Society Press.

[93] J. Hillston and L. Kloul. An efficient Kronecker representation for PEPA models. In *Proceedings of the Joint International Workshop, PAPM-PROBMIV 2001*, volume 2165 of *Lecture Notes in Computer Science*, pages 120–135. Springer-Verlag, 2001.

[94] J. Hillston and L. Kloul. Formal techniques for performance analysis: Blending SAN and PEPA. *Formal Aspects of Computing*, 19(1):3–33, 2007.

[95] J. Hillston and N. Thomas. Product form solution for a class of PEPA models. *Performance Evaluation*, 35(3–4):171–192, 1999.

[96] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[97] G.J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[98] W.C. Horrace. Some results on the multivariate truncated normal distribution. In *Journal of Multivariate Analysis*, volume 94, pages 209–221, 2005.

[99] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, second edition, 2004.

[100] P.A. Jacobson and E.D. Lazowska. Analyzing queueing networks with simultaneous resource possession. *Communications of the ACM*, 25(2):142–151, 1982.

[101] H.E. Jensen. Model checking probabilistic real time systems. In *Proceedings of the 7th Nordic Workshop on Programming Theory*, pages 247–261, Göteborg Sweden, 1996.

[102] B. Jonsson and K.G. Larsen. Specification and refinement of probabilistic processes. In *LICS '91: Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 266–277, Amsterdam, The Netherlands, 1991.

[103] G. Kahn. Natural semantics. In *STACS '87: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[104] J.-P. Katoen, D. Klink, and M.R. Neuhäußer. Compositional abstraction for stochastic systems. In *FORMATS '09: Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, pages 195–211, Budapest, Hungary, 2009. Springer-Verlag.

[105] J.-P. Katoen, D. Klink, M. Leucker, and V. Wolf. Three-valued abstraction for continuous-time Markov chains. In W. Damm and H. Hermanns, editors, *Proceedings of 19th International Conference on Computer-Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 316–329. Springer-Verlag, 2007.

[106] J.-P. Katoen, I.S. Zapreev, E.M. Hahn, H. Hermanns, and D.N. Jansen. The ins and outs of the probabilistic model checker MRMC. In *QEST '09: Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems*, pages 167–176. IEEE Computer Society, 2009.

[107] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. Abstraction refinement for probabilistic software. In N. Jones and M. Muller-Olm, editors, *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*, volume 5403 of *Lecture Notes in Computer Science*, pages 182–197. Springer-Verlag, 2009.

[108] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer-Verlag, 1976.

[109] L. Kleinrock. Analysis of a time-shared processor. 11:59–73, 1964.

[110] L. Kleinrock. *Queueing Systems, Volume I: Theory*. Wiley Interscience, New York, 1975.

[111] D.E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

[112] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[113] V. Kutsyy. com.kutsyy: A Java interface to latent variable spatial ordinal data (lvsdod), 2001. `http://www.kutsyy.com/java`.

[114] M.Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204, 2002.

[115] S.S. Lavenberg and M.S. Squillante. Performance evaluation in industry: A personal perspective. In *Performance Evaluation: Origins and Directions*, pages 3–13. Springer-Verlag, 2000.

[116] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioural Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[117] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Procedings of the 7th International Static Analysis Symposium (SAS 2000)*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[118] J.P. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to stochastic Petri nets: application to software performance engineering. *SIGSOFT Software Engineering Notes*, 29(1):25–36, 2004.

[119] M. B. Mamoun and N. Pekergin. Computing closed-form stochastic bounds on transient distributions of markov chains. In *SAINT-W '05: Proceedings of the 2005 Symposium on Applications and the Internet Workshops*, pages 260–263, Trento, Italy, 2005. IEEE Computer Society Press.

[120] M.A. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[121] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[122] M. Marzolla and S. Balsamo. UML-PSI: The UML performance simulator. In *QEST '04: Proceedings of the First International Conference on the Quantitative Evaluation of Systems*, pages 340–341, Sep 2004.

[123] K. Matthes. Zur theorie der bedienungsprozesse. In *Transactions of the Third Prague Conference on Information Theory and Statistical Decision Functions*, pages 513–528. Publishing House of the Czechoslovak Academy of Sciences, 1962.

[124] A. McIver and C. Morgan. Developing and reasoning about probabilistic programs in pGCL. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 123–155. Springer-Verlag, 2006.

[125] N. Medvidovic, A. Egyed, and D.S. Rosenblum. Round-trip software engineering using UML: From architecture to design and back. In *Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR '99)*, pages 1–8, Toulouse, France, 1999.

[126] V. Mertsiotakis. *Approximate Analysis Methods for Stochastic Process Algebras*. PhD thesis, Institut für Mathematische Maschinen und Datenverarbeitung, 1998.

[127] M.K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 31(9):913–917, 1982.

[128] D. Monniaux. Abstract interpretation of probabilistic semantics. In *Seventh International Static Analysis Symposium (SAS'00)*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. Springer-Verlag, 2000.

[129] R.E. Moore and F. Bierbaum. *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, 1979.

[130] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compiler Construction*, pages 213–228, 2002.

[131] G.C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, pages 128–139. ACM, Jan 2002.

[132] C.J. Neill and P.A. Laplante. Requirements engineering: The state of the practice. *IEEE Software*, 20(6):40–45, 2003.

[133] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[134] G. Norman. *Metric Semantics for Reactive Probabilistic Processes*. PhD thesis, School of Computer Science, University of Birmingham, 1997.

[135] OASIS. Web services business process execution language version 2.0. Apr 2007.

[136] P. O'Neil and E. O'Neil. *Database: Principles, Programming, and Performance*. Morgan Kaufmann, second edition, 2001.

[137] J.R.C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, 1995.

[138] D.B. Petriu and M. Woodside. A metamodel for generating performance models from UML designs. In T. Baar, A. Strohmeier, A. Moreira, and S.J. Mellor, editors, *7th International Conference on Modelling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2004.

[139] B.C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[140] A. Di Pierro, C. Hankin, and H. Wiklicky. Quantitative static analysis of distributed systems. *Journal of Functional Programming*, 15(5):703–749, 2005.

[141] A. Di Pierro and H. Wiklicky. Probabilistic abstract interpretation and statistical testing. In *PAPM-PROBMIV '02: Proceedings of the Second Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pages 211–212, Copenhagen, Denmark, 2002. Springer-Verlag.

[142] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. *SIGMETRICS Performance Evaluation Review*, 13(2):147–154, 1985.

[143] R. Pooley. Software engineering and performance: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 189–199, Limerick, Ireland, 2000. ACM.

[144] R. Pooley and P. King. The Unified Modelling Language and performance engineering. *IEE Proceedings: Software*, 146(1):2–10, Feb 1999.

[145] C. Priami. Stochastic π-calculus with general distributions. In M. Ribaudo, editor, *Proceedings of PAPM '96*. CLUT, 1996.

[146] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[147] M.O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.

[148] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[149] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, Copenhagen, Denmark, 2002. IEEE Computer Society.

[150] B. Ripley. *Stochastic Simulation*. John Wiley, New York, NY, USA, 1987.

[151] W.W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Monterey, California, USA, 1987. IEEE Computer Society Press.

[152] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[153] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, 1989.

[154] R. Schassberger. Insensitivity of steady-state distributions of generalized semi-Markov processes. part i. *The Annals of Probability*, 5(1):87–99, 1977.

[155] R. Schassberger. Insensitivity of steady-state distributions of generalized semi-Markov processes. part ii. *The Annals of Probability*, 6(1):85–93, 1978.

[156] A.L. Scherr. *An Analysis of Time-Shared Computer Systems*. PhD thesis, Massachusetts Institute of Technology, 1965.

[157] D.A. Schmidt. Abstract interpretation in the operational semantics hierarchy. Technical report, Kansas State University, 1997.

[158] P.J. Schweitzer. Aggregation methods for large Markov chains. In *Proceedings of the International Workshop on Computer Performance and Reliability*, pages 275–286. North-Holland Publishing Co., 1984.

[159] P.J. Schweitzer. A survey of aggregation-disaggregation in large Markov chains. In W.J. Stewart, editor, *Numerical Solution of Markov Chains*, pages 63–88. Marcel Dekker, 1991.

[160] J.P. Shen. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Education, 2004.

[161] H.A. Sholl and T.L. Booth. Software performance modeling using computation structures. *IEEE Transactions on Software Engineering*, 1(4):414–420, 1975.

[162] H. Simitci. *Storage Network Performance Analysis*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[163] H.A. Simon and A. Ando. Aggregation of variables in dynamic systems. *Econometrica*, 29(2):111–138, 1961.

[164] C.U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[165] C.U. Smith. Introduction to software performance engineering: Origins and outstanding problems. In *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 395–428. Springer, 2007.

[166] C.U. Smith and L.G. Williams. Software performance antipatterns. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 127–136, Ottawa, Ontario, Canada, 2000. ACM.

[167] C.U. Smith and L.G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, revised edition, 2003.

[168] J.E. Smith. A study of branch prediction strategies. In *ISCA '98: 25 years of the International Symposia on Computer Architecture (selected papers)*, pages 202–215, Barcelona, Spain, 1998. ACM.

[169] M.J.A. Smith. Stochastic modelling of communication protocols from source code. *Electronic Notes in Theoretical Computer Science*, 190(3):129–145, 2007.

[170] M.J.A. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science*, 220(3):43–59, 2008.

[171] I. Sommerville. *Software Engineering*. Pearson Education, eighth edition, 2007.

[172] W.R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[173] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[174] D. Stoyan. *Comparison Methods for Queues and Other Stochastic Models*. Wiley & Sons, New York, NY, USA, 1983.

[175] T. Suto, J.T. Bradley, and W.J. Knottenbelt. Performance trees: A new approach to quantitative performance specification. In *MASCOTS'06, 14th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 303–313. IEEE Computer Society Press, 2006.

[176] Y.L. Tong. *The Multivariate Normal Distribution*. Springer-Verlag, 1990.

[177] M. Tribastone. Bottom-up beats top-down hands down. In *Proceedings of the 6th Workshop on Process Algebra and Stochastically Timed Activities (PASTA)*, 2007. `http://pastaworkshop.org/2007/proceedings/`.

[178] M. Tribastone. The PEPA plug-in project. In M. Harchol-Balter, M. Kwiatkowska, and M. Telek, editors, *Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST)*, pages 53–54. IEEE Computer Society Press, 2007.

[179] M. Tribastone and S. Gilmore. Automatic translation of UML sequence diagrams into PEPA models. In *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 205–214, St Malo, France, 2008. IEEE Computer Society.

[180] F.I. Vokolos and E.J. Weyuker. Performance testing of software systems. In *WOSP '98: Proceedings of the 1st International Workshop on Software and Performance*, pages 80–87, Santa Fe, New Mexico, USA, 1998. ACM.

[181] M. Völter and T. Stahl. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, Inc., 2006.

[182] B. Wachter and L. Zhang. Best probabilistic transformers. In *VMCAI 2010: Proceedings of the 11th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 362–379. Springer-Verlag, 2010.

[183] M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-based development of service-oriented systems. *Formal Techniques for Networked and Distributed Systems - FORTE 2006*, pages 24–45, 2006.

[184] M. Woodside. From annotated software designs (UML SPT/MARTE) to model formalisms. In *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 429–467. Springer, 2007.

[185] G. Yaikhom, M. Cole, S. Gilmore, and J. Hillston. A structural approach for modelling performance of systems using skeletons. *Electronic Notes in Theoretical Computer Science*, 190(3):167–183, 2007.

[186] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

Appendix A

Extending Siril with Loops

In this appendix, we will extend the Siril language so that it contains loops, and discuss the problems that arise with our abstract interpretation in this context. We will show how to extend both the probabilistic semantics of Kozen and our probabilistic automaton semantics — both in the concrete and abstract cases. We will then discuss the issues that arise when we try to construct a memoised abstract interpretation that guarantees termination in the presence of loops. The work described here is based on that published in [170].

Let us begin by extending the syntax of commands, to include a `while`-loop:

$$C ::= \cdots \mid \texttt{while}\ B\ \mathsf{do}\ C$$

We can give this a denotational semantics as per Kozen [112], in terms of a least fixed point in the usual Scott-Strachey style. To construct this, we first define the following function, which takes an operator on measures, W, as its argument:

$$f(B,M)(W) = e_{[\![\neg B]\!]} + W \circ M \circ e_{[\![B]\!]}$$

Here, B is a boolean condition (hence the operator $e_{[\![B]\!]}$ is defined), and M is an operator on measures. We can define the semantics of a `while`-loop as the least fixed point of this function, with B and M corresponding to the loop condition and the body of the loop respectively:

$$[\![\texttt{while}\ B\ \mathsf{do}\ C]\!]_p = \mathrm{lfp}(f(B,[\![C]\!]_p))$$

The least fixed point operator computes the measure W such that $f(B,[\![C]\!]_p)(W) = W$. The proof that such a fixed point exists, and is unique, is given in [112].

We can also extend our probabilistic automaton semantics to deal with the `while`-loop. Intuitively, we can avoid the above fixed point construction by introducing an

```
O.f(X) {
  N := 10;
  Y := N−X;
  while (Y > 0) do
    X := X+1;
    Y := N−X;
  return X
}
```

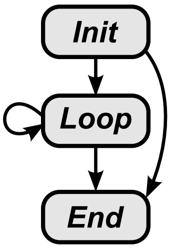


Figure A.1: An example Siril method with additive looping behaviour

additional stage into the automaton, corresponding to the entry point of the loop. Note that we will not give a label to this stage, as we did for those corresponding to remote procedure calls, since its behaviour is internal to the method. We can represent the looping behaviour by a cyclic transition relation, which re-enters the loop after computing its body, by returning to this stage. If we were to unroll the loop, we would effectively compute the fixed point, as done explicitly by Kozen. The probabilistic automaton semantics is as follows, where s^* is a fresh stage:

$$\begin{aligned}
[\![O.f \mid \texttt{while}\ B\ \texttt{do}\ C]\!]^S_{pa} &= [\![O.f \mid C]\!]^S_{pa} \cup \{s^*\}\\
[\![O.f \mid \texttt{while}\ B\ \texttt{do}\ C]\!]^T_{pa} &= \{\bigcirc \xrightarrow{e_{[\![B]\!]}} s^*\} \cup \{\bigcirc \xrightarrow{e_{[\![\neg B]\!]}} \bullet\} \cup\\
&\quad \{s^* \xrightarrow{M} s \mid \bigcirc \xrightarrow{M} s \in [\![O.f \mid C]\!]^T_{pa}\} \cup\\
&\quad \{s \xrightarrow{e_{[\![B]\!]}\circ M} s^* \mid s \xrightarrow{M} \bullet \in [\![O.f \mid C]\!]^T_{pa}\} \cup\\
&\quad \{s \xrightarrow{e_{[\![\neg B]\!]}\circ M} \bullet \mid s \xrightarrow{M} \bullet \in [\![O.f \mid C]\!]^T_{pa}\} \cup\\
&\quad \{s \xrightarrow{M} s' \in [\![O.f \mid C]\!]^T_{pa} \mid s \neq \bigcirc \wedge s' \neq \bullet\}
\end{aligned}$$

The abstract semantics $[\![O.f \mid \texttt{while}\ B\ \texttt{do}\ C]\!]^\sharp_{pa}$ is the same as the above, except that we replace $e_{[\![\cdot]\!]}$ with $e^\sharp_{[\![\cdot]\!]}$ and $[\![\cdot]\!]_{pa}$ with $[\![\cdot]\!]^\sharp_p$.

Up to this point, there are no difficulties with extending Siril with loops — both in terms of its concrete and abstract semantics. The problem comes when we try to *interpret* its abstract semantics, since there is no guarantee that it will terminate. Effectively, we can carry out the interpretation as presented in Section 4.4, but we would be unrolling the loop to construct all possible measures on all possible iterations, leading to an infinite state Markov chain.

As an example of this, Figure A.1 shows a small Siril program, alongside its control flow graph, which takes an input variable X, and increments it until it reaches the value 10. If the initial value of X is greater than 10, it exits immediately rather than entering the loop. This is a single-threaded program, and we only need to keep track of the state of three variables — X, Y and N. Note that *Init* corresponds to stage $\circ_{O.f}$ in the probabilistic automaton of the method $O.f$, and *End* corresponds to $\bullet_{O.f}$.

Before we can execute the abstract semantics, we need to define the initial distribution of the method's argument, X. For simplicity, let us assume that X initially has a mean of zero and a variance of one, and is not truncated — i.e. $X \sim T[\bot, \top]N_1(0,1)$. The initial measure over the method's variables is then the following truncated multivariate normal measure:

$$\begin{bmatrix} X \\ N \\ Y \end{bmatrix} \sim T\left[\begin{bmatrix} \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \end{bmatrix}\right] N_3\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\right)$$

This notation is an explicit expansion of the form $\boldsymbol{X} \sim T[\boldsymbol{a}, \boldsymbol{b}]N_{|\boldsymbol{X}|}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, for our case when $\boldsymbol{X} = [X, N, Y]^T$. The non-argument variables are initialised to have zero mean and variance, as per Equation 4.3.

After taking the first transition, from the *Init* to *Loop*, the measure is transformed into the following:

$$\begin{bmatrix} X \\ N \\ Y \end{bmatrix} \sim T\left[\begin{bmatrix} \bot \\ \bot \\ -0.5 \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \end{bmatrix}\right] N_3\left(\begin{bmatrix} 0 \\ 10 \\ 10 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}\right)$$

Three changes have occurred — N has been assigned the constant value 10, Y is now equal to $N - X$, and the condition $Y > 0$ has been applied. Recall that since the measure we describe is *continuous*, we treat an integer value a as corresponding to the interval $[a - 0.5, a + 0.5]$, hence the truncation of $[-0.5, \top]$ for the condition $Y > 0$. Note that $Cov(X,Y) = -Var(X) = -1$, capturing that Y is negatively correlated with X.

Although it is possible to go directly from the *Init* to *End*, the probability is so small that we will ignore that possibility — from an implementation point of view, this corresponds to discarding measures whose total weight is below a certain threshold ϵ. Strictly speaking, this is not a safe thing to do from the point of view of the abstract interpretation, but we can consider ϵ to be a parameter that determines the numerical accuracy of our technique.

We can now observe how the measure changes as we execute the body of the loop

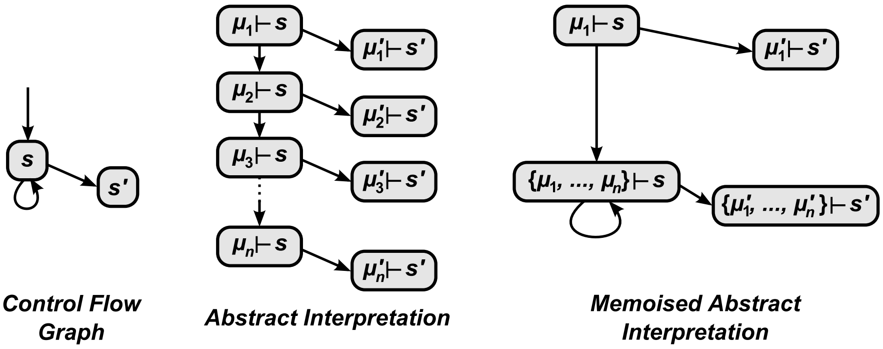


Figure A.2: Termination of a memoised abstract interpretation

multiple times:

$$
\begin{aligned}
\begin{bmatrix} X \\ N \\ Y \end{bmatrix} &\sim T\left\llbracket \begin{bmatrix} \bot \\ \bot \\ -0.5 \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \end{bmatrix} \right\rrbracket N_3\left(\begin{bmatrix} 0 \\ 10 \\ 10 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \right) \\
&\to T\left\llbracket \begin{bmatrix} \bot \\ \bot \\ -0.5 \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \end{bmatrix} \right\rrbracket N_3\left(\begin{bmatrix} 1 \\ 10 \\ 9 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \right) \\
&\to T\left\llbracket \begin{bmatrix} \bot \\ \bot \\ -0.5 \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \end{bmatrix} \right\rrbracket N_3\left(\begin{bmatrix} 2 \\ 10 \\ 8 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \right) \\
&\to T\left\llbracket \begin{bmatrix} \bot \\ \bot \\ -0.5 \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \end{bmatrix} \right\rrbracket N_3\left(\begin{bmatrix} 3 \\ 10 \\ 7 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \right) \\
&\to \ldots
\end{aligned}
$$

If we continue this way, we will completely unfold the loop, and explicitly construct the measure at each iteration. This is clearly undesirable for larger and more complex programs, as we cannot guarantee termination.

Rather than explicitly unfolding the loop like this, we would like instead to ‘predict’ the possible measures that can occur within the loop body — accepting that we will have to make a safe *over-approximation* for this to be possible in general. In other words, rather than associating a single measure with each stage in a Siril method, we can associate a *set* of measures with it. We can then construct a *widening operator* that allows us to over-approximate the set of measures that are possible at a given stage.

This idea is known as a *memoised abstract interpretation*, and is illustrated in Figure A.2. We can construct this in two stages:

1. Lift our abstract interpretation so that it operates over sets of measures ρ, rather than individual measures μ. This means that the transitions in the abstract interpretation will be as follows:

$$\rho \vdash s \to M(\rho) \vdash s' \quad \text{if } s \xrightarrow{M} s' \in [\![O.f]\!]^{T\sharp}_{pa}$$

   where $M(\rho) = \{ M(\mu) \mid \mu \in \rho \}$.

2. Construct a memoised abstract interpretation, which allows us to jump to an over-approximation of the above sets, rather than expanding them out completely:

$$\rho \vdash s \to^{\sharp} \mathit{Lookup}_{O.f}(s') \triangledown M(\rho) \vdash s' \quad \text{if } s \xrightarrow{M} s' \in [\![O.f]\!]^{T\sharp}_{pa}$$

   Here, $\mathit{Lookup}_{O.f} : [\![O.f]\!]^{S\sharp}_{p} \to \mathcal{P}(TM)$ is a map that records the set of measures that have been seen at a stage s during the abstract interpretation. We use TM to denote the set of all truncated multivariate normal measures. When we execute a transition $\rho \vdash s \to \rho' \vdash s'$ that computes a new set of measures for the stage s', we *combine* them with those already seen at that stage, using a widening operator $\triangledown : \mathcal{P}(TM) \times \mathcal{P}(TM) \to \mathcal{P}(TM)$, which over-approximates the union of two sets of measures. For looping behaviour, this allows us to jump to a value that encompasses all iterations of the loop — which in the worst case, might be the set of all measures, $\top = TM$.

   After each transition, we update the lookup function as follows:

$$\mathit{Lookup}_{O.f}(s') := \mathit{Lookup}_{O.f}(s') \triangledown M(\rho)$$

Whilst this general approach is standard in the abstract interpretation literature, there are particular problems when we consider our domain of truncated multivariate normal measures. The first problem is in choosing which sets of measures we can represent — i.e. the forms that ρ can take — and the second problem is in constructing the widening operator $\triangledown$, which combines these sets of measures.

If we return to the example from Figure A.1, we can see that at each iteration of the loop, the mean is altered by a constant offset, whereas the truncation intervals and the covariance matrix remain the same. We could therefore represent this series of measures more compactly by the following set:

$$\left\{ T\left[\begin{bmatrix} \bot \\ \bot \\ -0.5 \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \end{bmatrix}\right] N_3\left(\begin{bmatrix} 0 \\ 10 \\ 10 \end{bmatrix} + n\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}\right) \;\middle|\; n \in \mathbb{N}_{\geq 0} \right\}$$

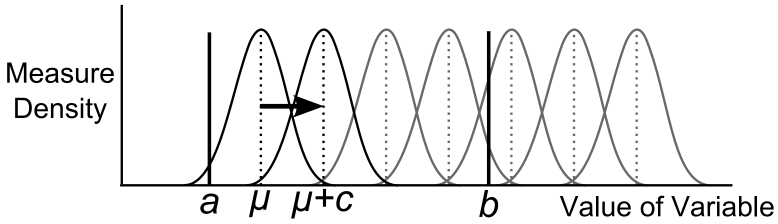


Figure A.3: A one-dimensional additive set of measures

This set corresponds to *all* the possible measures that can occur *within* the loop. If we sum the total weights of the measures of this set, we get an overall value of 9. This corresponds to us entering the loop nine times, as we would expect. Since the total weight after exiting the loop must be no greater than the total weight before entering it, we can conclude that the total weight of the measures in the *End* state is 1. This means that if we are in the *Loop* state, the probability of moving to the *End* state is $\frac{1}{9+1} = 0.1$, which corresponds to the expectation that there are ten iterations of the loop.

To represent such a set of measures in general, we could introduce the following parameterised form for ρ:

$$\{T[\boldsymbol{a},\boldsymbol{b}]N_N(\boldsymbol{\mu}+n\boldsymbol{c},\boldsymbol{\Sigma}) \mid n \in \mathbb{N}_{\geq 0}\}$$

This is illustrated in Figure A.3 in the case of a one-dimensional additive update. Note that although there are an infinite number of measures in the set, the sum of the measures truncated to the interval $[\boldsymbol{a},\boldsymbol{b}]$ has a finite total weight.

To illustrate how we could construct a widening operator $\triangledown$, let us consider how we might obtain the above parameterised set of measures from two singleton sets of measures. If $\mu_1 = T[\boldsymbol{a}_1,\boldsymbol{b}_1]N_N(\boldsymbol{\mu}_1,\boldsymbol{\Sigma}_1)$ and $\mu_2 = T[\boldsymbol{a}_2,\boldsymbol{b}_2]N_N(\boldsymbol{\mu}_2,\boldsymbol{\Sigma}_2)$, then:

$$\{\mu_1\}\triangledown\{\mu_2\} = \begin{cases} \{\mu_1\} & \text{if } \mu_1 = \mu_2 \\ \{T[\boldsymbol{a}',\boldsymbol{b}']N_N(\boldsymbol{\mu}_1+n\boldsymbol{c},\boldsymbol{\Sigma}_1) \mid n \in \mathbb{N}_{\geq 0}\} & \text{if } \boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 \\ \top & \text{otherwise} \end{cases}$$

where $\forall i.\ \boldsymbol{a}'(i) = \min\{\boldsymbol{a}_1(i),\boldsymbol{a}_2(i)\}$, $\forall i.\ \boldsymbol{b}'(i) = \max\{\boldsymbol{b}_1(i),\boldsymbol{b}_2(i)\}$, and $\boldsymbol{c} = \boldsymbol{\mu}_2 - \boldsymbol{\mu}_1$.

Whilst we can use this approach for certain simple classes of loop, such as the above, it quickly becomes very complicated when we try to generalise it. There are three main reasons for this:

1. It is hard to find sets of measures, such as the above, that classify the behaviour of more general classes of loop. Most loops have a more complex operation

that simply incrementing the value of a variable, and so we would need to find a *compact* way of representing the set of measures that this corresponds to. The problem becomes even more difficult when there is branching behaviour within the loop.

2. It is hard to construct widening operators that correctly identify the behaviour of a loop, just by looking at the measures on the state of the variables. For example, if we exchange the values of two variables, then we will be unable to detect this by looking at the measures if the variables have the same mean, variance, truncation interval, and covariances with the other variables. If they have different values, however, then we *can* tell this difference. This means that we have to be very careful when constructing a widening operator, and there are subtle considerations to make.

3. Possibly the most important problem is that we need to *manipulate* these sets of measures. When we exit a loop, we will subsequently perform operations on the variables that were modified within the loop. Consequently, we need to lift our measure operators to sets of measures. Given that these sets may be infinite, this is not easy to do, and leads to even more complicated parameterised forms for the sets of measures.

Because of these concerns, the main presentation of this thesis considered a version of the Siril language *without* looping behaviour. We believe that the correct approach to handling loops is not to follow the above presentation, but to make use of other techniques — perhaps, using a different analysis technique for the full language with loops, and using the abstract interpretation and collecting semantics presented in Chapter 4 to handle the non-looping fragment of the language. For a more detailed discussion of future work, see Chapter 8.

Appendix B

Abstract Interpretation of the Client-Server Example

Recall that the vector of variables for the example in Figure 3.1 is defined as follows:

$$\begin{array}{lllllllll}
\boldsymbol{X}(1) & = & \mathtt{quantity} & \boldsymbol{X}(2) & = & \mathtt{cash} & \boldsymbol{X}(3) & = & \mathtt{price} \\
\boldsymbol{X}(4) & = & \mathtt{success} & \boldsymbol{X}(5) & = & \mathtt{max\_order} & \boldsymbol{X}(6) & = & \langle \mathtt{price} - \mathtt{cash} \rangle \\
\boldsymbol{X}(7) & = & \langle \mathtt{quantity} - \mathtt{max\_order} \rangle & & & & & &
\end{array}$$

The measures indicated in the abstract interpretation of Figure 4.2 (b) are as follows:

$$\mu_1 = T\left(\begin{bmatrix} 0 \\ 0 \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{11} = T\left(\begin{bmatrix} 0 \\ 0 \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{111} = T\left(\begin{bmatrix} 10.5 \\ 0 \\ 84 \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 64 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 72 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 72 & 0 & 576 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{112} = T\left(\begin{bmatrix} 0.5 \\ 0 \\ 5 \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} 10.5 \\ \top \\ 105 \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 80 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 90 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 90 & 0 & 900 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{113} = T\left(\begin{bmatrix} 0 \\ 0 \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} 0.5 \\ \top \\ \top \\ \top \\ \top \\ \top \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{1111} = T\left(\begin{bmatrix} 10.5 \\ 0 \\ 84 \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} \top \\ \top \\ \top \\ \top \\ \top \\ 0.5 \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 64 \\ 0 \\ 0 \\ -6 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 72 & 0 & 0 & 72 & 0 \\ 0 & 4 & 0 & 0 & 0 & -4 & 0 \\ 72 & 0 & 576 & 0 & 0 & 576 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 72 & -4 & 576 & 0 & 0 & 580 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{1121} = T\left(\begin{bmatrix} 0.5 \\ 0 \\ 5 \\ \bot \\ \bot \\ -105 \\ \bot \end{bmatrix}, \begin{bmatrix} 10.5 \\ \top \\ 105 \\ \top \\ \top \\ 0.5 \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 80 \\ 0 \\ 0 \\ 10 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 90 & 0 & 0 & 90 & 0 \\ 0 & 4 & 0 & 0 & 0 & -4 & 0 \\ 90 & 0 & 900 & 0 & 0 & 900 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 90 & -4 & 900 & 0 & 0 & 904 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{1131} = T\left(\begin{bmatrix} 0 \\ 0 \\ \bot \\ \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}, \begin{bmatrix} 0.5 \\ \top \\ \top \\ \top \\ \top \\ 0.5 \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right)$$

$$\mu_{1112} = T\left(\begin{bmatrix}10.5\\0\\84\\\bot\\\bot\\0.5\\\bot\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\\top\\\top\\\top\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\64\\0\\0\\-6\\0\end{bmatrix},\begin{bmatrix}9&0&72&0&0&0&0\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&-4&576&0&0&580&0\\0&0&0&0&0&0&0\end{bmatrix}\right)$$

$$\mu_{1132} = T\left(\begin{bmatrix}0\\0\\\bot\\\bot\\\bot\\0.5\\\bot\end{bmatrix},\begin{bmatrix}0.5\\\top\\\top\\\top\\\top\\\top\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\0\\0\\0\\0\\0\end{bmatrix},\begin{bmatrix}9&0&0&0&0&0&0\\0&4&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{bmatrix}\right)$$

$$\mu_{11112} = T\left(\begin{bmatrix}10.5\\0\\84\\\bot\\\bot\\\bot\\0.5\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\\top\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\64\\0\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-4&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right)$$

$$\mu_{11212} = T\left(\begin{bmatrix}0.5\\0\\5\\\bot\\\bot\\-105\\0.5\end{bmatrix},\begin{bmatrix}10.5\\\top\\105\\\top\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\80\\0\\100\\10\\-92\end{bmatrix},\begin{bmatrix}9&0&90&0&0&90&9\\0&4&0&0&0&-4&0\\90&0&900&0&0&900&90\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\90&-4&900&0&0&904&90\\9&0&90&0&0&90&9\end{bmatrix}\right)$$

$$\mu_{11312} = T\left(\begin{bmatrix}0\\0\\\bot\\\bot\\\bot\\\bot\\0.5\end{bmatrix},\begin{bmatrix}0.5\\\top\\\top\\\top\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\0\\1\\100\\0\\-92\end{bmatrix},\begin{bmatrix}9&0&0&0&0&0&9\\0&4&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\9&0&0&0&0&0&9\end{bmatrix}\right)$$

$$\mu_{111121} = T\left(\begin{bmatrix}10.5\\\bot\\84\\0.5\\\bot\\\bot\\0.5\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\1.5\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\6\\64\\0\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&580&-576&0&0&-580&0\\72&-576&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-580&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right)$$

$$\mu_{112121} = T\left(\begin{bmatrix}0.5\\-105\\5\\0.5\\\bot\\-105\\0.5\end{bmatrix},\begin{bmatrix}10.5\\\top\\105\\1.5\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\-10\\80\\0\\100\\10\\-92\end{bmatrix},\begin{bmatrix}9&0&90&0&0&90&9\\0&904&-900&0&0&-904&0\\90&-900&900&0&0&900&90\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\90&-904&900&0&0&904&90\\9&0&90&0&0&90&9\end{bmatrix}\right)$$

$$\mu_{113121} = T\left(\begin{bmatrix}0\\0\\\bot\\0.5\\\bot\\\bot\\0.5\end{bmatrix},\begin{bmatrix}0.5\\\top\\\top\\1.5\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\0\\1\\100\\0\\-92\end{bmatrix},\begin{bmatrix}9&0&0&0&0&0&9\\0&4&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\9&0&0&0&0&0&9\end{bmatrix}\right)$$

$$\mu_{1122} = T\left(\begin{bmatrix}0.5\\0\\5\\\bot\\\bot\\0.5\\\bot\end{bmatrix},\begin{bmatrix}10.5\\\top\\105\\\top\\\top\\\top\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\80\\0\\0\\10\\0\end{bmatrix},\begin{bmatrix}9&0&90&0&0&0&0\\0&4&0&0&0&-4&0\\90&0&900&0&0&900&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&-4&900&0&0&904&0\\0&0&0&0&0&0&0\end{bmatrix}\right)$$

$$\mu_{11111} = T\left(\begin{bmatrix}10.5\\0\\84\\\bot\\\bot\\\bot\\\bot\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\\top\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\64\\1\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-4&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right)$$

$$\mu_{11211} = T\left(\begin{bmatrix}0.5\\0\\5\\\bot\\\bot\\-105\\\bot\end{bmatrix},\begin{bmatrix}10.5\\\top\\105\\\top\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\80\\1\\100\\10\\-92\end{bmatrix},\begin{bmatrix}9&0&90&0&0&90&9\\0&4&0&0&0&-4&0\\90&0&900&0&0&900&90\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\90&-4&900&0&0&904&90\\9&0&90&0&0&90&9\end{bmatrix}\right)$$

$$\mu_{11311} = T\left(\begin{bmatrix}0\\0\\\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix},\begin{bmatrix}0.5\\\top\\\top\\\top\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\0\\1\\100\\0\\-92\end{bmatrix},\begin{bmatrix}9&0&0&0&0&0&9\\0&4&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\9&0&0&0&0&0&9\end{bmatrix}\right)$$

$$\mu_{111111} = T\left(\begin{bmatrix}10.5\\\bot\\84\\0.5\\\bot\\\bot\\\bot\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\1.5\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\6\\64\\1\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&580&-576&0&0&-580&0\\72&-576&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-580&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right)$$

$$\mu_{112111} = T\left(\begin{bmatrix}0.5\\-105\\5\\0.5\\\bot\\-105\\\bot\end{bmatrix},\begin{bmatrix}10.5\\\top\\105\\1.5\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\-10\\80\\1\\100\\10\\-92\end{bmatrix},\begin{bmatrix}9&0&90&0&0&90&9\\0&904&-900&0&0&-904&0\\90&-900&900&0&0&900&90\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\90&-904&900&0&0&904&90\\9&0&90&0&0&90&9\end{bmatrix}\right)$$

$$\mu_{113111} = T\left(\begin{bmatrix}0\\0\\\bot\\0.5\\\bot\\\bot\\\bot\end{bmatrix},\begin{bmatrix}0.5\\\top\\\top\\1.5\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\0\\1\\100\\0\\-92\end{bmatrix},\begin{bmatrix}9&0&0&0&0&0&9\\0&4&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\9&0&0&0&0&0&9\end{bmatrix}\right)$$

$$\mu_{111112} = \left\{T\left(\begin{bmatrix}10.5\\0\\84\\\bot\\\bot\\\bot\\\bot\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\0.5\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\64\\1\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-4&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right),T\left(\begin{bmatrix}10.5\\0\\84\\1.5\\\bot\\\bot\\\bot\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\\top\\\top\\0.5\\0.5\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\64\\1\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-4&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right)\right\}$$

$$\mu_{111122} = \left\{T\left(\begin{bmatrix}10.5\\0\\84\\\bot\\\bot\\\bot\\0.5\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\0.5\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\64\\0\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-4&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right),T\left(\begin{bmatrix}10.5\\0\\84\\1.5\\\bot\\\bot\\0.5\end{bmatrix},\begin{bmatrix}\top\\\top\\\top\\\top\\\top\\0.5\\\top\end{bmatrix}\right)N_7\left(\begin{bmatrix}8\\70\\64\\0\\100\\-6\\-92\end{bmatrix},\begin{bmatrix}9&0&72&0&0&72&9\\0&4&0&0&0&-4&0\\72&0&576&0&0&576&72\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\72&-4&576&0&0&580&72\\9&0&72&0&0&72&9\end{bmatrix}\right)\right\}$$

$$\mu_{112112} = \left\{ T\left(\begin{bmatrix} 0.5 \\ 0 \\ 5 \\ \perp \\ \perp \\ -105 \\ \perp \end{bmatrix}, \begin{bmatrix} 10.5 \\ \top \\ 105 \\ 0.5 \\ \top \\ 0.5 \\ 0.5 \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 80 \\ 1 \\ 100 \\ 10 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 90 & 0 & 0 & 90 & 9 \\ 0 & 4 & 0 & 0 & 0 & -4 & 0 \\ 90 & 0 & 900 & 0 & 0 & 900 & 90 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 90 & -4 & 900 & 0 & 0 & 904 & 90 \\ 9 & 0 & 90 & 0 & 0 & 90 & 9 \end{bmatrix}\right), T\left(\begin{bmatrix} 0.5 \\ 0 \\ 5 \\ 1.5 \\ \perp \\ -105 \\ \perp \end{bmatrix}, \begin{bmatrix} 10.5 \\ \top \\ 105 \\ \top \\ \top \\ 0.5 \\ 0.5 \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 80 \\ 1 \\ 100 \\ 10 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 90 & 0 & 0 & 90 & 9 \\ 0 & 4 & 0 & 0 & 0 & -4 & 0 \\ 90 & 0 & 900 & 0 & 0 & 900 & 90 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 90 & -4 & 900 & 0 & 0 & 904 & 90 \\ 9 & 0 & 90 & 0 & 0 & 90 & 9 \end{bmatrix}\right) \right\}$$

$$\mu_{112122} = \left\{ T\left(\begin{bmatrix} 0.5 \\ 0 \\ 5 \\ \perp \\ \perp \\ -105 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 10.5 \\ \top \\ 105 \\ 0.5 \\ \top \\ 0.5 \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 80 \\ 0 \\ 100 \\ 10 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 90 & 0 & 0 & 90 & 9 \\ 0 & 4 & 0 & 0 & 0 & -4 & 0 \\ 90 & 0 & 900 & 0 & 0 & 900 & 90 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 90 & -4 & 900 & 0 & 0 & 904 & 90 \\ 9 & 0 & 90 & 0 & 0 & 90 & 9 \end{bmatrix}\right), T\left(\begin{bmatrix} 0.5 \\ 0 \\ 5 \\ 1.5 \\ \perp \\ -105 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 10.5 \\ \top \\ 105 \\ \top \\ \top \\ 0.5 \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 80 \\ 0 \\ 100 \\ 10 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 90 & 0 & 0 & 90 & 9 \\ 0 & 4 & 0 & 0 & 0 & -4 & 0 \\ 90 & 0 & 900 & 0 & 0 & 900 & 90 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 90 & -4 & 900 & 0 & 0 & 904 & 90 \\ 9 & 0 & 90 & 0 & 0 & 90 & 9 \end{bmatrix}\right) \right\}$$

$$\mu_{113112} = \left\{ T\left(\begin{bmatrix} 0 \\ 0 \\ \perp \\ \perp \\ \perp \\ \perp \\ \perp \end{bmatrix}, \begin{bmatrix} 0.5 \\ \top \\ \top \\ 0.5 \\ \top \\ 0.5 \\ 0.5 \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 1 \\ 100 \\ 0 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 9 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 & 9 \end{bmatrix}\right), T\left(\begin{bmatrix} 0 \\ 0 \\ \perp \\ 1.5 \\ \perp \\ \perp \\ \perp \end{bmatrix}, \begin{bmatrix} 0.5 \\ \top \\ \top \\ \top \\ \top \\ 0.5 \\ 0.5 \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 1 \\ 100 \\ 0 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 9 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 & 9 \end{bmatrix}\right) \right\}$$

$$\mu_{113122} = \left\{ T\left(\begin{bmatrix} 0 \\ 0 \\ \perp \\ \perp \\ \perp \\ \perp \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.5 \\ \top \\ \top \\ 0.5 \\ \top \\ 0.5 \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 1 \\ 100 \\ 0 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 9 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 & 9 \end{bmatrix}\right), T\left(\begin{bmatrix} 0 \\ 0 \\ \perp \\ 1.5 \\ \perp \\ \perp \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.5 \\ \top \\ \top \\ \top \\ \top \\ 0.5 \\ \top \end{bmatrix}\right) N_7\left(\begin{bmatrix} 8 \\ 70 \\ 0 \\ 1 \\ 100 \\ 0 \\ -92 \end{bmatrix}, \begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 9 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 & 9 \end{bmatrix}\right) \right\}$$

The total measures are as follows, where we write '≈ 0' to mean that the total measure is zero to within seven decimal places, as opposed to being precisely zero:

Measure	Total Measure	Measure	Total Measure	Measure	Total Measure
μ_1	0.996170	μ_{1132}	0	μ_{111121}	≈ 0
μ_{11}	0.996170	μ_{11111}	0.202414	μ_{112121}	≈ 0
μ_{111}	0.202328	μ_{11211}	0.369921	μ_{113121}	≈ 0
μ_{112}	0.791462	μ_{11311}	0.002421	μ_{111112}	0
μ_{113}	0.002379	μ_{11112}	≈ 0	μ_{112112}	0
μ_{1111}	0.202411	μ_{11212}	≈ 0	μ_{113112}	0
μ_{1121}	0.369831	μ_{11312}	≈ 0	μ_{111122}	0
μ_{1131}	0.002379	μ_{111111}	0.202414	μ_{112122}	0
μ_{1112}	≈ 0	μ_{112111}	0.369921	μ_{113122}	0
μ_{1122}	0.421657	μ_{113111}	0.002421		

Appendix C

Kronecker Operators

The Kronecker operators are a class of tensor operator, allowing two matrices to be combined in such a way that their original elements can be recovered — the operators are universal. If each matrix describes a probabilistic transition system over a given state space, then the Kronecker product describes a probabilistic transition system over the Cartesian product of the state spaces.

Definition C.1. *The* Kronecker product *of an $m \times n$ matrix $\boldsymbol{A}$ and a $p \times q$ matrix $\boldsymbol{B}$ is an $mp \times nq$ matrix $\boldsymbol{A} \otimes \boldsymbol{B}$ defined as follows:*

$$\begin{aligned}
\boldsymbol{A}\otimes\boldsymbol{B} &= \begin{bmatrix} a_{11}\boldsymbol{B} & \dots & a_{1n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & \dots & a_{mn}\boldsymbol{B} \end{bmatrix} \\
&= \begin{bmatrix}
a_{11}b_{11} & \dots & a_{11}b_{1q} & \dots & a_{1n}b_{11} & \dots & a_{1n}b_{1q} \\
\vdots & \ddots & \vdots & & \vdots & \ddots & \vdots \\
a_{11}b_{p1} & \dots & a_{11}b_{pq} & \dots & a_{1n}b_{p1} & \dots & a_{1n}b_{pq} \\
\vdots & & \vdots & \ddots & \vdots & & \vdots \\
a_{m1}b_{11} & \dots & a_{m1}b_{1q} & \dots & a_{mn}b_{11} & \dots & a_{mn}b_{1q} \\
\vdots & \ddots & \vdots & & \vdots & \ddots & \vdots \\
a_{m1}b_{p1} & \dots & a_{m1}b_{pq} & \dots & a_{mn}b_{p1} & \dots & a_{mn}b_{pq}
\end{bmatrix}
\end{aligned}$$

If $\boldsymbol{A}$ and $\boldsymbol{B}$ are stochastic matrices, then $\boldsymbol{A} \otimes \boldsymbol{B}$ is also stochastic, with the transition probabilities encoding the simultaneous firing of transitions in $\boldsymbol{A}$ and $\boldsymbol{B}$.

Using the above definition, we can also define a Kronecker notion of summation. This is only defined for square matrices.

Definition C.2. *The* Kronecker sum *of an* $n \times n$ *matrix* $\boldsymbol{A}$ *and a* $m \times m$ *matrix* $\boldsymbol{B}$ *is an* $mn \times mn$ *matrix* $\boldsymbol{A} \oplus \boldsymbol{B}$ *defined as follows:*

$$\boldsymbol{A} \oplus \boldsymbol{B} \quad = \quad \boldsymbol{A} \otimes \boldsymbol{I}_m + \boldsymbol{I}_n \otimes \boldsymbol{B}$$

where $\boldsymbol{I}_n$ *denotes the* $n \times n$ *identity matrix.*

If $\boldsymbol{A}$ and $\boldsymbol{B}$ are stochastic matrices, then $\boldsymbol{A} \oplus \boldsymbol{B}$ corresponds to the *interleaving* of the the DTMCs that they describe. Unlike the Kronecker product, the Kronecker sum only allows transitions to take place independently — either a transition from $\boldsymbol{A}$ can fire, or a transition from $\boldsymbol{B}$, but not from both simultaneously.

Appendix D

Proofs for Chapter 6

Theorem 6.1.1. *Consider two generator matrices $\boldsymbol{Q}_1 = (r_1, \boldsymbol{P}_1)$ and $\boldsymbol{Q}_2 = (r_2, \boldsymbol{P}_2)$, corresponding to the same state space S ($\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$ are both $|S| \times |S|$ matrices). Then $\boldsymbol{Q}_1 + \boldsymbol{Q}_2$ can be written as follows:*

$$\boldsymbol{Q}_1 + \boldsymbol{Q}_2 = (r_1, \boldsymbol{P}_1) + (r_2, \boldsymbol{P}_2) = \left(r_1 + r_2, \frac{r_1}{r_1 + r_2} \boldsymbol{P}_1 + \frac{r_2}{r_1 + r_2} \boldsymbol{P}_2 \right)$$

where $(r_1 + r_2)(s) = r_1(s) + r_2(s)$, and $\frac{r_i}{r_1+r_2}(s) = \frac{r_i(s)}{r_1(s)+r_2(s)}$, $i \in \{1, 2\}$, for all $s \in S$.

Proof: The proof is as follows. For an entry s, s', when $s \neq s'$, in $\boldsymbol{Q}_1 + \boldsymbol{Q}_2$, we have:

$$\begin{aligned}
(\boldsymbol{Q}_1 + \boldsymbol{Q}_2)(s, s') &= ((r_1, \boldsymbol{P}_1) + (r_2, \boldsymbol{P}_2))(s, s') \\
&= (r_1, \boldsymbol{P}_1)(s, s') + (r_2, \boldsymbol{P}_2)(s, s') \\
&= r_1(s)\boldsymbol{P}_1(s, s') + r_2(s)\boldsymbol{P}_2(s, s') \\
&= (r_1(s) + r_2(s))\left(\tfrac{r_1(s)}{r_1(s)+r_2(s)}\boldsymbol{P}_1(s, s') + \tfrac{r_2(s)}{r_1(s)+r_2(s)}\boldsymbol{P}_2(s, s')\right) \\
&= \left(r_1 + r_2, \tfrac{r_1}{r_1+r_2}\boldsymbol{P}_1 + \tfrac{r_2}{r_1+r_2}\boldsymbol{P}_2\right)(s, s')
\end{aligned}$$

When $s = s'$, we similarly have:

$$\begin{aligned}
(\boldsymbol{Q}_1 + \boldsymbol{Q}_2)(s, s) &= ((r_1, \boldsymbol{P}_1) + (r_2, \boldsymbol{P}_2))(s, s) \\
&= (r_1, \boldsymbol{P}_1)(s, s) + (r_2, \boldsymbol{P}_2)(s, s) \\
&= r_1(s)(\boldsymbol{P}_1(s, s) - 1) + r_2(s)(\boldsymbol{P}_2(s, s) - 1) \\
&= (r_1(s) + r_2(s))\left(\tfrac{r_1(s)}{r_1(s)+r_2(s)}\boldsymbol{P}_1(s, s) + \tfrac{r_2(s)}{r_1(s)+r_2(s)}\boldsymbol{P}_2(s, s) - 1\right) \\
&= \left(r_1 + r_2, \tfrac{r_1}{r_1+r_2}\boldsymbol{P}_1 + \tfrac{r_2}{r_1+r_2}\boldsymbol{P}_2\right)(s, s)
\end{aligned}$$

■

Theorem 6.1.3. *For all well-formed PEPA models C, the CTMC induced by the semantics of PEPA and the CTMC described by the generator matrix $\boldsymbol{Q}(C)$, projected onto the derivative set* $\mathrm{ds}(C)$ *(the reachable state space of C), are isomorphic.*

Proof: Since, in a PEPA model, the transitions of each action type are independent from one another, we can consider them separately. We therefore need to prove that for each action type, the transition rates induced by the Kronecker form are identical to those induced by the PEPA semantics, as given in [91].

We proceed by induction on the structure of the system equation. In the base case, for a sequential component C_i, $\boldsymbol{Q}_a(C_i) = (r_{i,a}, \boldsymbol{P}_{i,a})$ corresponds, by Definition 6.1.2, precisely to those activities of type a that C_i can perform. In other words, for $s, s' \in S_i$, $r_{i,a}(s)$ is the apparent rate of action type a in state s, and $\boldsymbol{P}_{i,a}(s,s')$ is the relative probability of moving to state s', if we perform an a activity in state s.

For the inductive case, we make the hypothesis that there is a transition $C_1 \xrightarrow{(a,r)} C_2$ induced by the PEPA semantics of a component C — where C may be a composition of sequential components, and $C_1, C_2 \in \mathrm{ds}(C)$ — if and only if $\boldsymbol{Q}_a(C)(C_1,C_2) = r_a(C_1)\boldsymbol{P}_a(C_1,C_2) = r$. We can ignore the case when $C_1 = C_2$ as self loops cancel out in the generator matrix.

Assume that there is an additional component C' such that $C'_1 \xrightarrow{(a,r')} C'_2$ iff $\boldsymbol{Q}_a(C')(C'_1,C'_2) = r'$, as above. We will prove that for all sets of action types L, $C \underset{L}{\bowtie} C'$ induces a transition $C_1 \underset{L}{\bowtie} C'_1 \xrightarrow{(a,R)} C_2 \underset{L}{\bowtie} C'_2$ iff $\boldsymbol{Q}_a(C \underset{L}{\bowtie} C')(C_1 \underset{L}{\bowtie} C'_1, C_2 \underset{L}{\bowtie} C'_2) = R$.

Consider the case $a \in L$. Then:

$$\begin{aligned}
& \boldsymbol{Q}_a(C \underset{L}{\bowtie} C')(C_1 \underset{L}{\bowtie} C'_1, C_2 \underset{L}{\bowtie} C'_2) \\
= \ & (\boldsymbol{Q}_a(C) \circledast \boldsymbol{Q}_a(C'))(C_1 \underset{L}{\bowtie} C'_1, C_2 \underset{L}{\bowtie} C'_2) \\
= \ & ((r_a, \boldsymbol{P}_a) \circledast (r'_a, \boldsymbol{P}'_a))(C_1 \underset{L}{\bowtie} C'_1, C_2 \underset{L}{\bowtie} C'_2) \\
= \ & (\min\{r_a, r'_a\}, \boldsymbol{P}_a \otimes \boldsymbol{P}'_a)(C_1 \underset{L}{\bowtie} C'_1, C_2 \underset{L}{\bowtie} C'_2) \\
= \ & \min\{r_a(C_1), r'_a(C'_1)\}(\boldsymbol{P}_a(C_1,C_2) \times \boldsymbol{P}'_a(C'_1,C'_2)) \\
= \ & \min\{r_a(C_1), r'_a(C'_1)\}\frac{r}{r_a(C_1)}\frac{r'}{r'_a(C'_1)}
\end{aligned}$$

where the final step follows from the induction hypothesis. This is equal by definition to the PEPA semantics of cooperation for action types $a \in L$, hence gives the rate R of the transition $C_1 \underset{L}{\bowtie} C'_1 \xrightarrow{(a,R)} C_2 \underset{L}{\bowtie} C'_2$ induced by the PEPA semantics.

Consider the case $a \notin L$. Then:

$$\begin{aligned}
& \boldsymbol{Q}_a(C \underset{L}{\bowtie} C')(C_1 \underset{L}{\bowtie} C_1', C_2 \underset{L}{\bowtie} C_2') \\
= \; & (\boldsymbol{Q}_a(C) \odot \boldsymbol{Q}_a(C'))(C_1 \underset{L}{\bowtie} C_1', C_2 \underset{L}{\bowtie} C_2') \\
= \; & ((r_a, \boldsymbol{P}_a) \odot (r_a', \boldsymbol{P}_a'))(C_1 \underset{L}{\bowtie} C_1', C_2 \underset{L}{\bowtie} C_2') \\
= \; & ((r_a, \boldsymbol{P}_a) \circledast (r_\top, \boldsymbol{I}) + (r_\top, \boldsymbol{I}) \circledast (r_a', \boldsymbol{P}_a'))(C_1 \underset{L}{\bowtie} C_1', C_2 \underset{L}{\bowtie} C_2') \\
= \; & \min\{r_a, \top\}(C_1, C_1')\boldsymbol{P}_a(C_1, C_2)\boldsymbol{I}(C_1', C_2') + \min\{\top, r_a'\}(C_1, C_1')\boldsymbol{I}(C_1, C_2)\boldsymbol{P}_a'(C_1', C_2') \\
= \; & r_a(C_1)\boldsymbol{P}_a(C_1, C_2)\boldsymbol{I}(C_1', C_2') + r_a'(C_1')\boldsymbol{I}(C_1, C_2)\boldsymbol{P}_a'(C_1', C_2') \\
= \; & \begin{cases} r_a(C_1)\boldsymbol{P}_a(C_1, C_2) & \text{if } C_2' = C_1' \\ r_a'(C_1')\boldsymbol{P}_a'(C_1', C_2') & \text{if } C_2 = C_1 \\ 0 & \text{otherwise} \end{cases} \\
= \; & \begin{cases} r & \text{if } C_2' = C_1' \\ r' & \text{if } C_2 = C_1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

where the final step follows from the induction hypothesis. This corresponds to the PEPA semantics of cooperation for action types $a \notin L$, where the activities of the two components take place independently. In other words, $C_1 \underset{L}{\bowtie} C_1' \xrightarrow{(a,r)} C_2 \underset{L}{\bowtie} C_1'$ if $C_1 \xrightarrow{(a,r)} C_2$ in component C, and $C_1 \underset{L}{\bowtie} C_1' \xrightarrow{(a,r')} C_1 \underset{L}{\bowtie} C_2'$ if $C_1' \xrightarrow{(a,r')} C_2'$ in C'. ■

Theorem 6.3.4. *Consider a CTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$. For any uniformisation constant $\lambda \geq \max_{s \in S} r(s)$, and any abstraction $(S^\sharp, \alpha)$ on $\mathcal{M}$, the following holds:*

$$Abs_{(S^\sharp,\alpha)}(Unif_\lambda(\mathcal{M})) \ \leq \ ACTMC_\lambda\Big(AbsComp_{(S^\sharp,\alpha)}(\mathcal{M})\Big)$$

Proof: Consider a CTMC $\mathcal{M} = (S, \boldsymbol{\pi}^{(0)}, \boldsymbol{P}, r, L)$, which is not necessarily uniform. Let us define its uniformisation with respect to λ as:

$$\overline{\mathcal{M}} = Unif_\lambda(\mathcal{M}) = (S, \boldsymbol{\pi}^{(0)}, \overline{\boldsymbol{P}}, \overline{r}, L)$$

where $\overline{\boldsymbol{P}}$ is the uniformised probability transition matrix (see Definition 5.1.2). Since $\overline{\mathcal{M}}$ is a uniformised CTMC, we can define its abstraction $\mathcal{M}^\sharp$ with respect to $(S^\sharp, \alpha)$ as follows:

$$\mathcal{M}^\sharp = Abs_{(S^\sharp,\alpha)}(\overline{\mathcal{M}}) = (S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^L, \boldsymbol{P}^U, \lambda, L^\sharp)$$

where $\boldsymbol{P}^L$ and $\boldsymbol{P}^U$ give the lower and upper bounds for the uniformised transition probabilities, and $L^\sharp$ is the abstract labelling function (see Definition 5.6.3).

Considering the abstract CTMC component, let us define:

$$\mathcal{M}^{\sharp\sharp} = AbsComp_{(S^\sharp,\alpha)}(\mathcal{M}) = (S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}^L_C, \boldsymbol{P}^U_C, r^L_C, r^U_C, L^\sharp)$$

where $\boldsymbol{P}^L_C$, $\boldsymbol{P}^U_C$, r^L_C and r^U_C give the lower and upper bounds for the transition probabilities and exit rates, and $L^\sharp$ is the abstract labelling function (see Definition 6.3.3). The CTMC induced by $\mathcal{M}^{\sharp\sharp}$ is given by:

$$ACTMC_\lambda(\mathcal{M}^{\sharp\sharp}) = (S^\sharp, \boldsymbol{\pi}^{(0)\sharp}, \boldsymbol{P}'^L, \boldsymbol{P}'^U, \lambda, L^\sharp)$$

where $\boldsymbol{P}'^L$ and $\boldsymbol{P}'^U$ give upper and lower bounds for the uniformised transition probabilities, after converting the abstract CTMC component into an abstract CTMC (see Definition 6.3.2).

Let us now consider the lower bounding matrices — we require that for all $s, s' \in S^\sharp$, $\boldsymbol{P}'^L(s,s') \leq \boldsymbol{P}^L(s,s')$. Consider first the case when $s \neq s'$. Then we have:

$$\begin{array}{rcll}
\boldsymbol{P}'^L(s,s') & = & \dfrac{r^L_C(s)}{\lambda}\boldsymbol{P}^L_C(s,s') & \text{Definition 6.3.2} \\
 & = & \dfrac{1}{\lambda}\min\limits_{t\in\gamma(s)} r(t) \min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s')} \boldsymbol{P}(t,t') & \text{Definition 6.3.3} \\
 & \leq & \min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s')} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t') & \text{Since } \min(ab) \geq \min(a)\min(b) \\
 & = & \min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s')} \overline{\boldsymbol{P}}(t,t') & \text{Definition 5.1.2} \\
 & = & \boldsymbol{P}^L(s,s') & \text{Definition 5.6.3}
\end{array}$$

Note that the central step works on the basis that all rates and probabilities are positive, hence the minimum of the product is greater than or equal to the product of the minima.

For the case when $s = s'$, we have:

$$\begin{array}{rcll}
\boldsymbol{P}'^{L}(s,s) & = & 1 - \dfrac{r_C^U(s)}{\lambda} + \dfrac{r_C^L(s)}{\lambda}\boldsymbol{P}_C^L(s,s) & \text{Definition 6.3.2} \\
& = & 1 - \dfrac{1}{\lambda}\max\limits_{t\in\gamma(s)} r(t) + \dfrac{1}{\lambda}\min\limits_{t\in\gamma(s)} r(t) \min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s)} \boldsymbol{P}(t,t') & \text{Definition 6.3.3} \\
& \leq & 1 - \dfrac{1}{\lambda}\max\limits_{t\in\gamma(s)} r(t) + \min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s)} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t') & \\
& \leq & 1 - \max\limits_{t\in\gamma(s)} \sum\limits_{t'\in S\setminus\{t\}} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t') + \min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s)\setminus\{t\}} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t') & \text{See Below} \\
& = & \min\limits_{t\in\gamma(s)} \left(\left(1 - \sum\limits_{t'\in S\setminus\{t\}} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t') \right) + \sum\limits_{t'\in\gamma(s)\setminus\{t\}} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t') \right) & \\
& = & \min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s)} \overline{\boldsymbol{P}}(t,t') & \text{Definition 5.1.2} \\
& = & \boldsymbol{P}^L(s,s) & \text{Definition 5.6.3}
\end{array}$$

To prove the noted step, let us assume that we have the minimum and maximum values, $t_1^{\min}$, $t_2^{\min}$, $t_3^{\max}$ and $t_4^{\min}$, of the following sums:

- $t_1^{\max}$ maximises: $\max\limits_{t\in\gamma(s)}(r(t))$.
- $t_2^{\min}$ minimises: $\min\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s)} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t')$.
- $t_3^{\max}$ maximises: $\max\limits_{t\in\gamma(s)} \sum\limits_{t'\in S\setminus\{t\}} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t')$.
- $t_4^{\min}$ minimises: $\max\limits_{t\in\gamma(s)} \sum\limits_{t'\in\gamma(s)\setminus\{t\}} \dfrac{r(t)}{\lambda}\boldsymbol{P}(t,t')$.

Using these minimising and maximising values, we have:

$$\begin{aligned}
& 1-\frac{r(t_1^{\max})}{\lambda} && + \sum_{t'\in\gamma(s)} \frac{r(t_2^{\min})}{\lambda}\boldsymbol{P}(t_2^{\min},t') \\
\leq\ & 1-\frac{r(t_1^{\max})}{\lambda} && + \sum_{t'\in\gamma(s)} \frac{r(t_4^{\min})}{\lambda}\boldsymbol{P}(t_4^{\min},t') \\
=\ & 1-\frac{r(t_1^{\max})}{\lambda}+\frac{r(t_4^{\min})}{\lambda}\boldsymbol{P}(t_4^{\min},t_4^{\min}) && + \sum_{t'\in\gamma(s)\setminus\{t_4^{\min}\}} \frac{r(t_4^{\min})}{\lambda}\boldsymbol{P}(t_4^{\min},t') \\
\leq\ & 1-\frac{r(t_3^{\max})}{\lambda}+\frac{r(t_3^{\max})}{\lambda}\boldsymbol{P}(t_3^{\max},t_3^{\max}) && + \sum_{t'\in\gamma(s)\setminus\{t_4^{\min}\}} \frac{r(t_4^{\min})}{\lambda}\boldsymbol{P}(t_4^{\min},t') \\
=\ & 1-\frac{r(t_3^{\max})}{\lambda}(1-\boldsymbol{P}(t_3^{\max},t_3^{\max})) && + \sum_{t'\in\gamma(s)\setminus\{t_4^{\min}\}} \frac{r(t_4^{\min})}{\lambda}\boldsymbol{P}(t_4^{\min},t') \\
=\ & 1-\frac{r(t_3^{\max})}{\lambda}\sum_{t'\in S\setminus\{t_3^{\max}\}}\boldsymbol{P}(t_3^{\max},t') && + \sum_{t'\in\gamma(s)\setminus\{t_4^{\min}\}} \frac{r(t_4^{\min})}{\lambda}\boldsymbol{P}(t_4^{\min},t') \\
=\ & 1-\sum_{t'\in S\setminus\{t_3^{\max}\}}\frac{r(t_3^{\max})}{\lambda}\boldsymbol{P}(t_3^{\max},t') && + \sum_{t'\in\gamma(s)\setminus\{t_4^{\min}\}} \frac{r(t_4^{\min})}{\lambda}\boldsymbol{P}(t_4^{\min},t')
\end{aligned}$$

Hence it holds that $\boldsymbol{P'}^{L}(s,s') \leq \boldsymbol{P}^{L}(s,s')$.

By a similar argument, we can show that $\boldsymbol{P}^{U}(s,s') \leq \boldsymbol{P'}^{U}(s,s')$, hence we have $\mathcal{M}^{\sharp} \leq ACTMC_{\lambda}(\mathcal{M}^{\sharp\sharp})$. ■

Theorem 6.3.5. *Consider two PEPA components C_1 and C_2, with abstractions $(S^\sharp_1, \alpha_1)$ and $(S^\sharp_2, \alpha_2)$ respectively. Let $\mathcal{M}^{\sharp\sharp}_{i,a} = AbsComp_{(S^\sharp_i,\alpha_i)}(\boldsymbol{Q}_a(C_i))$ for $i \in \{1,2\}$. Then for all λ such that $Unif_\lambda(C_1 \underset{L}{\bowtie} C_2)$ is defined, the following holds:*

$$Abs_{(S^\sharp,\alpha)}\left(Unif_\lambda\left(\boldsymbol{Q}\left(C_1 \underset{L}{\bowtie} C_2\right)\right)\right) \leq ACTMC_\lambda\left(\sum_{a\in L}\mathcal{M}^{\sharp\sharp}_{1,a} \circledast \mathcal{M}^{\sharp\sharp}_{2,a} + \sum_{a\in\overline{L}}\mathcal{M}^{\sharp\sharp}_{1,a} \odot \mathcal{M}^{\sharp\sharp}_{2,a}\right)$$

where $S^\sharp = S^\sharp_1 \times S^\sharp_2$, $\alpha(s_1,s_2) = (\alpha_1(s_1),\alpha_2(s_2))$, and $\overline{L} = (\mathcal{A}ct(C_1) \cup \mathcal{A}ct(C_2)) \setminus L$.

Proof: Consider first a particular action type $a \in L \cup \overline{L}$. This results in the following term from the above comparison (expanding out the Kronecker operator on the left hand side):

$$Abs_{(S^\sharp,\alpha)}(Unif_\lambda(\boldsymbol{Q}_a(C_1) \oplus \boldsymbol{Q}_a(C_2))) \leq ACTMC_\lambda\left(\mathcal{M}^{\sharp\sharp}_{1,a} \oplus \mathcal{M}^{\sharp\sharp}_{2,a}\right)$$

Where $\oplus = \circledast$ if $a \in L$, and $\odot$ if $a \in \overline{L}$.

From Theorem 6.3.4, we have the following:

$$Abs_{(S^\sharp,\alpha)}(Unif_\lambda(\boldsymbol{Q}_a(C_1) \oplus \boldsymbol{Q}_a(C_2))) \leq ACTMC_\lambda\left(AbsComp_{(S^\sharp,\alpha)}(\boldsymbol{Q}_a(C_1) \oplus \boldsymbol{Q}_a(C_2))\right)$$

We therefore need to show that:

$$AbsComp_{(S^\sharp,\alpha)}(\boldsymbol{Q}_a(C_1) \oplus \boldsymbol{Q}_a(C_2)) = \mathcal{M}^{\sharp\sharp}_{1,a} \oplus \mathcal{M}^{\sharp\sharp}_{2,a}$$

Consider the case when $a \in L$, and therefore $\oplus = \circledast$. We have:

$$\boldsymbol{Q}_a(C_1) \circledast \boldsymbol{Q}_a(C_2) = (S_1 \times S_2, \boldsymbol{\pi}^{(0)}_1 \otimes \boldsymbol{\pi}^{(0)}_2, \boldsymbol{P}_{1,a} \otimes \boldsymbol{P}_{2,a}, \min\{r_{1,a}, r_{2,a}\}, L_1 \times L_2)$$

The abstract CTMC component $\mathcal{M}^{\sharp\sharp}_a$ that this induces is as follows:

$$\begin{aligned}\mathcal{M}^{\sharp\sharp}_a &= AbsComp_{(S^\sharp,\alpha)}(\boldsymbol{Q}_a(C_1) \circledast \boldsymbol{Q}_a(C_2)) \\ &= \big(S^\sharp_1 \times S^\sharp_2, \boldsymbol{\pi}^{(0)\sharp}_1 \otimes \boldsymbol{\pi}^{(0)\sharp}_2, (\boldsymbol{P}_{1,a} \otimes \boldsymbol{P}_{2,a})^L, (\boldsymbol{P}_{1,a} \otimes \boldsymbol{P}_{2,a})^U, \\ &\qquad (\min\{r_{1,a}, r_{2,a}\})^L, (\min\{r_{1,a}, r_{2,a}\})^U, L^\sharp_1 \times L^\sharp_2\big)\end{aligned}$$

But notice that the lower bound $(\boldsymbol{P}_{1,a} \otimes \boldsymbol{P}_{2,a})^L$ is the same as $\boldsymbol{P}^L_{1,a} \otimes \boldsymbol{P}^L_{2,a}$, since the minimum of a product is the same as the product of the minima, for positive values. Furthermore, the lower bound for the rate function, $(\min\{r_{1,a}, r_{2,a}\})^L$, is the same as taking the minimum of the lower bounding rate functions, $\min\{r^L_{1,a}, r^L_{2,a}\}$. The same holds for the upper bounds. But this is the same as the composition of the abstract CTMC components of C_1 and C_2 for action type a:

$$\mathcal{M}^{\sharp\sharp}_{1,a} \circledast \mathcal{M}^{\sharp\sharp}_{2,a} = \left(S^\sharp_1 \times S^\sharp_2, \boldsymbol{\pi}^{(0)\sharp}_1 \otimes \boldsymbol{\pi}^{(0)\sharp}_2, \boldsymbol{P}^L_{1,a} \otimes \boldsymbol{P}^L_{2,a}, \boldsymbol{P}^U_{1,a} \otimes \boldsymbol{P}^U_{2,a}, \min\{r^L_{1,a}, r^L_{2,a}\}, \min\{r^U_{1,a}, r^U_{2,a}\}, L^\sharp_1 \times L^\sharp_2\right)$$

It therefore follows that $\mathcal{M}_a^{\sharp\sharp} = \mathcal{M}_{1,a}^{\sharp\sharp} \circledast \mathcal{M}_{2,a}^{\sharp\sharp}$.

Consider the case when $a \in \overline{L}$, and therefore $\obar = \odot$. We have:

$$\boldsymbol{Q}_a(C_1) \odot \boldsymbol{Q}_a(C_2) = (S_1 \times S_2, \boldsymbol{\pi}_1^{(0)} \otimes \boldsymbol{\pi}_2^{(0)}, \boldsymbol{P}_{1,a} \oplus \boldsymbol{P}_{2,a}, r_{1,a} + r_{2,a}, L_1 \times L_2)$$

This induces the following abstract CTMC component:

$$\begin{aligned}
\mathcal{M}_a^{\sharp\sharp} &= AbsComp_{(S^\sharp,\alpha)}(\boldsymbol{Q}_a(C_1) \odot \boldsymbol{Q}_a(C_2)) \\
&= \Big(S_1^\sharp \times S_2^\sharp, \boldsymbol{\pi}_1^{(0)\sharp} \otimes \boldsymbol{\pi}_2^{(0)\sharp}, (\boldsymbol{P}_{1,a} \oplus \boldsymbol{P}_{2,a})^L, (\boldsymbol{P}_{1,a} \oplus \boldsymbol{P}_{2,a})^U, \\
&\qquad (r_{1,a} + r_{2,a})^L, (r_{1,a} + r_{2,a})^U, L_1^\sharp \times L_2^\sharp\Big)
\end{aligned}$$

But we have, for:

$$\begin{aligned}
(r_{1,a} + r_{2,a})^L(s_1^\sharp, s_2^\sharp) &= \min_{s_1 \in \gamma(s_1^\sharp), s_2 \in \gamma(s_2^\sharp)} r_{1,a}(s_1) + r_{2,a}(s_2) \\
&= \min_{s_1 \in \gamma(s_1^\sharp)} r_{1,a}(s_1) + \min_{s_2 \in \gamma(s_2^\sharp)} r_{2,a}(s_2) \\
&= r_{1,a}^L(s_1^\sharp) + r_{2,a}^L(s_2^\sharp)
\end{aligned}$$

The same follows for the upper bound of the rate function, and we follow a similar argument for the bounds of the probabilistic transition matrices. Hence this is the same as the composition of the abstract CTMC components of C_1 and C_2 for action type a:

$$\mathcal{M}_{1,a}^{\sharp\sharp} \oplus \mathcal{M}_{2,a}^{\sharp\sharp} = \Big(S_1^\sharp \times S_2^\sharp, \boldsymbol{\pi}_1^{(0)\sharp} \otimes \boldsymbol{\pi}_2^{(0)\sharp}, \boldsymbol{P}_{1,a}^L \oplus \boldsymbol{P}_{2,a}^L, \boldsymbol{P}_{1,a}^U \oplus \boldsymbol{P}_{2,a}^U, r_{1,a}^L + r_{2,a}^L, r_{1,a}^U + r_{2,a}^U, L_1^\sharp \times L_2^\sharp\Big)$$

Hence $\mathcal{M}_a^{\sharp\sharp} = \mathcal{M}_{1,a}^{\sharp\sharp} \circledast \mathcal{M}_{2,a}^{\sharp\sharp}$.

We have shown that the safety of the abstraction is preserved for all action types $a \in L \cup \overline{L}$, and so it follows that this also holds for the sum over all action types. ■

Theorem 6.4.5. *For generator matrices* $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ *and* $\boldsymbol{Q}'_a = r'_a(\boldsymbol{P}'_a - \boldsymbol{I}')$*, if* $\boldsymbol{Q}_a \leq_{\mathrm{rst}} \boldsymbol{Q}'_a$ *and for all* $s \in S$*,* $r_a(s) \leq r'_a(s)$*, then* $\boldsymbol{Q}_a \leq_{\mathrm{st}} \boldsymbol{Q}'_a$*.*

Proof: For any λ that is not exceeded in magnitude by any diagonal element of $\boldsymbol{Q}_a$ or $\boldsymbol{Q}'_a$, we need to show that $\frac{\boldsymbol{Q}_a}{\lambda} + \boldsymbol{I} \leq_{\mathrm{st}} \frac{\boldsymbol{Q}'_a}{\lambda} + \boldsymbol{I}$, by the definition of the stochastic ordering on CTMCs. This corresponds to showing that:

$$\frac{r_a \boldsymbol{P}_a}{\lambda} + \left(1 - \frac{r_a}{\lambda}\right)\boldsymbol{I} \leq_{\mathrm{st}} \frac{r'_a \boldsymbol{P}'_a}{\lambda} + \left(1 - \frac{r'_a}{\lambda}\right)\boldsymbol{I}$$

remembering that r_a and r'_a are apparent rate functions, which can be written as vectors. By the definition of the strong stochastic ordering, this requires that, for each row s and for all states s':

$$\frac{r_a(s)}{\lambda}\sum_{t \succ s'} \boldsymbol{P}_a(s,t) + \left(1 - \frac{r_a(s)}{\lambda}\right)\mathbf{1}_{s' \prec s} \leq \frac{r'_a(s)}{\lambda}\sum_{t \succ s'} \boldsymbol{P}'_a(s,t) + \left(1 - \frac{r'_a(s)}{\lambda}\right)\mathbf{1}_{s' \prec s}$$

where $\mathbf{1}_{s' \prec s}$ is the indicator function, evaluating to one if the condition $s' \prec s$ holds, and to zero otherwise. If we are above the diagonal element (i.e. the indicator term evaluates to zero), then the relation holds since $r_a(s) \leq r'_a(s)$ and $\boldsymbol{P}_a \leq_{\mathrm{st}} \boldsymbol{P}'_a$. Otherwise, we have, for $s' \prec s$:

$$\frac{r_a(s)}{\lambda}\sum_{t \succ s'} \boldsymbol{P}_a(s,t) - \frac{r_a(s)}{\lambda} \leq \frac{r'_a(s)}{\lambda}\sum_{t \succ s'} \boldsymbol{P}'_a(s,t) - \frac{r'_a(s)}{\lambda}$$

which, on re-arranging, gives:

$$\frac{r'_a(s)}{r_a(s)} \leq \frac{1 - \sum_{t \succ s'} \boldsymbol{P}_a(s,t)}{1 - \sum_{t \succ s'} \boldsymbol{P}'_a(s,t)}$$

But since we know that the left-hand side is less than or equal to the minimum of all possible ratios on the right-hand side, this holds for all s'. ■

Theorem 6.4.6. *For a generator matrix $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$, if $\boldsymbol{Q}_a$ is rate-wise monotone, and for all $s \prec s' \in S$, $r_a(s) \leq r_a(s')$, then $\boldsymbol{Q}_a$ is monotone.*

Proof: For any λ that is not exceeded in magnitude by any diagonal element of $\boldsymbol{Q}_a$, we need to show that $\frac{\boldsymbol{Q}_a}{\lambda} + \boldsymbol{I}$ is monotone, by the definition of monotonicity for CTMCs. This corresponds to showing that:

$$\frac{r_a \boldsymbol{P}_a}{\lambda} + \left(1 - \frac{r_a}{\lambda}\right)\boldsymbol{I}$$

is monotone, where we write the apparent rate function r_a as a vector. By the definition of monotonicity, we require for all states s and s', such that $s \prec s'$ (two rows that we compare), and for all states s'' (elements along the row):

$$\frac{r_a(s)}{\lambda} \sum_{t \succ s''} \boldsymbol{P}_a(s,t) + \left(1 - \frac{r_a(s)}{\lambda}\right)\mathbf{1}_{s'' \prec s} \leq \frac{r_a(s')}{\lambda} \sum_{t \succ s''} \boldsymbol{P}_a(s',t) + \left(1 - \frac{r_a(s')}{\lambda}\right)\mathbf{1}_{s'' \prec s'}$$

If we are above the diagonal element in both rows (i.e. the indicator terms $\mathbf{1}_{s'' \prec s}$ and $\mathbf{1}_{s'' \prec s'}$ both evaluate to zero), then the relation holds since $r_a(s) \leq r_a(s')$ and $\boldsymbol{P}_a$ is monotone. Otherwise, we have to consider two cases: when $s \prec s'' \prec s'$, and when $s'' \prec s$.

When $s \prec s'' \prec s'$, we have:

$$\frac{r_a(s)}{\lambda} \sum_{t \succ s''} \boldsymbol{P}_a(s,t) \leq \frac{r_a(s')}{\lambda} \sum_{t \succ s''} \boldsymbol{P}_a(s',t) + 1 - \frac{r_a(s')}{\lambda}$$

which holds as before, since $1 - \frac{r_a(s')}{\lambda} > 0$.

Finally, when $s'' \prec s$, we have:

$$\frac{r_a(s)}{\lambda} \sum_{t \succ s''} \boldsymbol{P}_a(s,t) - \frac{r_a(s)}{\lambda} \leq \frac{r_a(s')}{\lambda} \sum_{t \succ s''} \boldsymbol{P}_a(s',t) - \frac{r_a(s')}{\lambda}$$

which, on re-arranging, gives:

$$\frac{r_a(s')}{r_a(s)} \leq \frac{1 - \sum_{t \succ s''} \boldsymbol{P}_a(s,t)}{1 - \sum_{t \succ s''} \boldsymbol{P}_a(s',t)}$$

But since we know that the left-hand side is less than or equal to the minimum of all possible ratios on the right-hand side, this holds for all s''. ■

Theorem 6.4.12 (Monotonicity). *Let two components, C_1 and C_2, occur in contexts ©$_1$ and ©$_2$ respectively, such that $C_1 \in$ ©$_2$ and $C_2 \in$ ©$_1$. Let ©$_1$ be internally bounded by B^1_{int}, and ©$_2$ by B^2_{int}, for action type a.*

If the matrices $\boldsymbol{Q}_{1,a} = r_{1,a}(\boldsymbol{P}_{1,a} - \boldsymbol{I})$ of C_1 and $\boldsymbol{Q}_{2,a} = r_{2,a}(\boldsymbol{P}_{2,a} - \boldsymbol{I})$ of C_2 are context-bounded rate-wise monotone by B^1_{int} and B^2_{int} respectively, then $(r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a})$ is context-bounded rate-wise monotone by the internal bound B^3_{int} of the context ©$_3$ of $C_1 \underset{L}{\bowtie} C_2$, for all action sets L.

To prove this theorem, we will first establish the following two lemmas. Lemma D.1 shows that the Kronecker product preserves monotonicity, and Lemma D.2 shows that the minimum of two monotone functions is also monotone. We omit the subscript a for clarity, hence we write $\boldsymbol{P}_i$ in place of $\boldsymbol{P}_{i,a}$, and r_i in place of $r_{i,a}$ for $i \in \{1,2\}$:

Lemma D.1. *Let $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ be monotone stochastic matrices describing the PEPA components C_1 and C_2 respectively, which have state spaces $(S_1, \prec_1)$ and $(S_2, \prec_2)$. Then $\boldsymbol{P}_1 \otimes \boldsymbol{P}_2$ is also monotone under the lifted orderings $\prec^L_1$ and $\prec^L_2$ on $S_1 \times S_2$.*

Proof: Consider states $(s_1, s_2) \prec^L_1 (s'_1, s'_2)$, recalling that this implies that $s_1 \prec_1 s'_1$ and $\neg\exists s''_2.\ s''_2 \prec_2 s_2$. We need to show that the following inequality holds, for all $s \in \mathrm{ds}(C_1)$ and $t \in \mathrm{ds}(C_2)$:

$$\sum_{(s',t') \succ^L_1 (s,t)} \boldsymbol{P}_1(s_1, s')\boldsymbol{P}_2(s_2, t') \leq \sum_{(s',t') \succ^L_1 (s,t)} \boldsymbol{P}_1(s'_1, s')\boldsymbol{P}_2(s'_2, t')$$

But in order for there to be any states $(s', t') \succ (s, t)$, t must be the smallest state in $(S_2, \prec_2)$. Hence this is equivalent to:

$$\sum_{s' \succ_1 s} \sum_{t' \in S_2} \boldsymbol{P}_1(s_1, s')\boldsymbol{P}_2(s_2, t') \leq \sum_{s' \succ_1 s} \sum_{t' \in S_2} \boldsymbol{P}_1(s'_1, s')\boldsymbol{P}_2(s'_2, t')$$

which we rewrite to give:

$$\sum_{s' \succ_1 s} \boldsymbol{P}_1(s_1, s') \leq \sum_{s' \succ_1 s} \boldsymbol{P}_1(s'_1, s')$$

This holds since $\boldsymbol{P}_1$ is monotone under $(S_1, \prec_1)$. The proof of monotonicity under $(S_1 \times S_2, \prec^L_2)$ follows similarly. ■

Lemma D.2. *Let r_1 and r_2 be monotone functions. Then $r_3(s_1, s_2) = \min\{r_1(s_1), r_2(s_2)\}$ is also monotone, under the orderings $(S_1 \times S_2, \prec^L_1)$ and $(S_1 \times S_2, \prec^L_2)$.*

Proof: Consider states $(s_1,s_2) \prec_1^L (s_1',s_2')$. By definition, $s_1 \prec_1 s_1'$ and either $s_2 \prec_1 s_2'$ or $s_2 = s_2'$. There are two cases to consider:

Case 1: $\min\{r_1(s_1'),r_2(s_2')\} = r_1(s_1')$. Then:

$$\begin{aligned}\min\{r_1(s_1),r_2(s_2)\} &\leq r_1(s_1)\\ &\leq r_1(s_1')\\ &\leq \min\{r_1(s_1'),r_2(s_2')\}\end{aligned}$$

Case 2: $\min\{r_1(s_1'),r_2(s_2')\} = r_2(s_2')$. Then:

$$\begin{aligned}\min\{r_1(s_1),r_2(s_2)\} &\leq r_2(s_2)\\ &\leq r_2(s_2')\\ &\leq \min\{r_1(s_1'),r_2(s_2')\}\end{aligned}$$

Hence r_3 is monotone with respect to $(S_1\times S_2,\prec_1^L)$. The proof of monotonicity under $(S_1\times S_2,\prec_2^L)$ follows similarly. ■

Proof: [Theorem 6.4.12] Let $(S_1,\prec_1)$ and $(S_2,\prec_2)$ be the state spaces of components C_1 and C_2 respectively. We will show that the generator matrix $\min\{r_1,r_2\}(\boldsymbol{P}_1\otimes\boldsymbol{P}_2-\boldsymbol{I})$ is monotone with respect to $(S_1\times S_2,\prec_1^L)$.

We know from Lemma D.1 that the matrix $\boldsymbol{P}_1\otimes\boldsymbol{P}_2$ is monotone, and from Lemma D.2 that the apparent rate function $\min\{r_1,r_2\}$ is monotone increasing. Hence, for all states $(s_1,s_2) \prec_1^L (s_1',s_2')$, we need to show that:

$$\max\left\{B_{int}^3,\frac{\min\{r_1(s_1'),r_2(s_2')\}}{\min\{r_1(s_1),r_2(s_2)\}}\right\} \leq \min_{(t_1,t_2)\prec_1^L(s_1,s_2)}\left\{\frac{1-\sum\limits_{(t_1',t_2')\succ_1^L(t_1,t_2)}\boldsymbol{P}_1(s_1,t_1')\boldsymbol{P}_2(s_2,t_2')}{1-\sum\limits_{(t_1',t_2')\succ_1^L(t_1,t_2)}\boldsymbol{P}_1(s_1',t_1')\boldsymbol{P}_2(s_2',t_2')}\right\}$$

where B_{int}^3 is the internal bound of the context ©″ of $C \underset{L}{\bowtie} C'$:

Let (t_1,t_2) be the state under which the ratio on the right hand side is at a minimum. Since $t_1 \prec_1 s_1$ by definition of $\prec_1^L$, we know that the following relation holds:

$$\max\left\{B_{int}^1,\frac{r_1(s_1')}{r_1(s_1)}\right\} \leq \frac{1-\sum\limits_{t_1'\succ_1 t_1}\boldsymbol{P}_1(s_1,t_1')}{1-\sum\limits_{t_1'\succ_1 t_1}\boldsymbol{P}_1(s_1',t_1')}$$

Furthermore, since by definition $B_{int}^1 \geq \frac{r_2(s_2')}{r_2(s_2)}$, $B_{int}^2 \geq \frac{r_1(s_1')}{r_1(s_1)}$, and $B_{int}^3 \leq \min\{B_{int}^1,B_{int}^2\}$, we can infer that:

$$\max\left\{B_{int}^3,\frac{r_1(s_1')}{r_1(s_1)},\frac{r_2(s_2')}{r_2(s_2)}\right\} \leq \max\left\{B_{int}^1,\frac{r_1(s_1')}{r_1(s_1)}\right\}$$

To complete the proof, we need to make use of the following observation:

Observation D.3. *For all positive $a, b, c, d \in \mathbb{R}$:*

$$\max\left\{\frac{a}{b}, \frac{c}{d}\right\} \geq \frac{\min\{a, c\}}{\min\{b, d\}}$$

since $\frac{a}{b} \geq \frac{\min\{a,c\}}{b}$ and $\frac{c}{d} \geq \frac{\min\{a,c\}}{d}$.

Using this observation, and the fact that t_2 must be the smallest state in $(S_2, <_2)$ by the definition of $<^L_1$:

$$\begin{aligned}
\max\left\{B^3_{int}, \frac{\min\{r_1(s'_1), r_2(s'_2)\}}{\min\{r_1(s_1), r_2(s_2)\}}\right\} &\leq \max\left\{B^3_{int}, \frac{r_1(s'_1)}{r_1(s_1)}, \frac{r_2(s'_2)}{r_2(s_2)}\right\} \\
&\leq \max\left\{B^1_{int}, \frac{r_1(s'_1)}{r_1(s_1)}\right\} \\
&\leq \frac{1 - \sum\limits_{t'_1 >_1 t_1} \boldsymbol{P}_1(s_1, t'_1)}{1 - \sum\limits_{t'_1 >_1 t_1} \boldsymbol{P}_1(s'_1, t'_1)} \\
&= \frac{1 - \sum\limits_{t'_1 >_1 t_1} \boldsymbol{P}_1(s_1, t'_1) \sum\limits_{t'_2 \in S_2} \boldsymbol{P}_2(s_2, t'_2)}{1 - \sum\limits_{t'_1 >_1 t_1} \boldsymbol{P}_1(s'_1, t'_1) \sum\limits_{t'_2 \in S_2} \boldsymbol{P}_2(s'_2, t'_2)} \\
&\leq \frac{1 - \sum\limits_{(t'_1, t'_2) >^L_1 (t_1, t_2)} \boldsymbol{P}_1(s_1, t'_1) \boldsymbol{P}_2(s_2, t'_2)}{1 - \sum\limits_{(t'_1, t'_2) >^L_1 (t_1, t_2)} \boldsymbol{P}_1(s'_1, t'_1) \boldsymbol{P}_2(s'_2, t'_2)} \\
&= \min_{(t_1, t_2) < (s_1, s_2)} \left\{ \frac{1 - \sum\limits_{(t'_1, t'_2) >^L_1 (t_1, t_2)} \boldsymbol{P}_1(s_1, t'_1) \boldsymbol{P}_2(s_2, t'_2)}{1 - \sum\limits_{(t'_1, t'_2) >^L_1 (t_1, t_2)} \boldsymbol{P}_1(s'_1, t'_1) \boldsymbol{P}_2(s'_2, t'_2)} \right\}
\end{aligned}$$

The proof of monotonicity under $(S_1 \times S_2, <^L_2)$ follows similarly.

Thus context-bounded rate-wise monotonicity is preserved by ⊛. ■

Theorem 6.4.13 (Lumpability). *Let C_1 and C_2 be PEPA models with generator matrices $\boldsymbol{Q}_1 = \sum_a \boldsymbol{Q}_{1,a}$ and $\boldsymbol{Q}_2 = \sum_a \boldsymbol{Q}_{2,a}$, where $\boldsymbol{Q}_{1,a} = r_{1,a}(\boldsymbol{P}_{1,a} - \boldsymbol{I})$ and $\boldsymbol{Q}_{2,a} = r_{2,a}(\boldsymbol{P}'_{2,a} - \boldsymbol{I})$. Then for all action types a, if the terms $\boldsymbol{Q}_{1,a}$ in $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_{2,a}$ in $\boldsymbol{Q}_2$ are ordinarily lumpable according to the partitions $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively, then the term $\boldsymbol{Q}_a = (r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a})$ in $\boldsymbol{Q} = \sum_a \boldsymbol{Q}_a$ is ordinarily lumpable according to $\mathcal{L}_1 \times \mathcal{L}_2$.*

Proof: Observe that $\boldsymbol{Q}$ is the generator matrix of $C_1 \underset{L}{\bowtie} C_2$, for some cooperation set L. For each action type a, $\boldsymbol{Q}_{1,a}$ is strongly equivalent to the lumped matrix $\mathcal{L}_1(\boldsymbol{Q}_{1,a})$, in that any a activity in $\boldsymbol{Q}_{1,a}$ can be matched by an a activity in $\mathcal{L}_1(\boldsymbol{Q}_{1,a})$. Similarly, $\boldsymbol{Q}_{2,a}$ and $\mathcal{L}_2(\boldsymbol{Q}_{2,a})$ are strongly equivalent.

It has been proven in [91] that the PEPA cooperation combinator preserves strong equivalence, and that strong equivalence implies ordinary lumpability. It follows that $\boldsymbol{Q}_a = (r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a})$ is strongly equivalent to:

$$\mathcal{L}_1(r_{1,a}, \boldsymbol{P}_{1,a}) \circledast \mathcal{L}_2(r_{2,a}, \boldsymbol{P}_{2,a}) = (\mathcal{L}_1 \times \mathcal{L}_2)((r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a}))$$

Hence $\boldsymbol{Q}_a$ is ordinarily lumpable according to the partition $\mathcal{L}_1 \times \mathcal{L}_2$.

Since for all action types $\boldsymbol{Q}_a$ is ordinarily lumpable, it follows that $\boldsymbol{Q} = \sum_a \boldsymbol{Q}_a$ is ordinarily lumpable according to $\mathcal{L}_1 \times \mathcal{L}_2$. Therefore, the operator $\circledast$ preserves ordinary lumpability over all action types. ■

Theorem 6.4.14 (Stochastic Order). *Consider the components C_i and C'_i, with generator matrices $\boldsymbol{Q}_{i,a} = r_{i,a}(\boldsymbol{P}_{i,a} - \boldsymbol{I})$ and $\boldsymbol{Q}'_{i,a} = r'_{i,a}(\boldsymbol{P}'_{i,a} - \boldsymbol{I})$, for $i \in \{1,2\}$ and action type a. Let $\boldsymbol{Q}_{i,a} \overset{B^i_{comp}}{\leq_{\mathrm{rst}}} \boldsymbol{Q}'_{i,a}$, with contexts $\copyright_i \leq_{\mathrm{st}} \copyright'_i$, where B^i_{comp} is the comparative bound of $\copyright_i$ and $\copyright'_i$. If B^3_{comp} is the comparative bound of the contexts $\copyright_1 \cap \copyright_2$ and $\copyright'_1 \cap \copyright'_2$, we have $(r_{1,a}, \boldsymbol{P}_{1,a}) \circledast (r_{2,a}, \boldsymbol{P}_{2,a}) \overset{B^3_{comp}}{\leq_{\mathrm{rst}}} (r'_{1,a}, \boldsymbol{P}'_{1,a}) \circledast (r'_{2,a}, \boldsymbol{P}'_{2,a})$.*

Proof: For clarity, we will omit the subscript a, and hence write $\boldsymbol{P}_i$ in place of $\boldsymbol{P}_{i,a}$, and r_i in place of $r_{i,a}$ for $i \in \{1,2\}$. We omit the proofs that $\boldsymbol{P}_1 \otimes \boldsymbol{P}_2 \leq_{\mathrm{st}} \boldsymbol{P}'_1 \otimes \boldsymbol{P}'_2$ and that $\min\{r_1, r_2\}(s_1, s_2) \leq \min\{r'_1, r'_2\}(s_1, s_2)$ for all $(s_1, s_2) \in S_1 \times S_2$, since they are very similar to Lemma D.1 and Lemma D.2 from the proof of Theorem 6.4.12 (Montonicity).

Let $(S_1, \prec_1)$ and $(S_2, \prec_2)$ be the state spaces of components C_1, C'_1 and C_2, C'_2 respectively. We will show that the generator matrix $\min\{r'_1, r'_2\}(\boldsymbol{P}'_1 \otimes \boldsymbol{P}'_2 - \boldsymbol{I})$ is a context-bounded rate-wise upper bound of $\min\{r_1, r_2\}(\boldsymbol{P}_1 \otimes \boldsymbol{P}_2 - \boldsymbol{I})$, with respect to the ordering $(S_1 \times S_2, \prec^L_1)$.

We need to show that the following inequality holds, for all states (s_1, s_2):

$$\max\left\{ B^3_{comp}, \frac{\min\{r'_1(s_1), r'_2(s_2)\}}{\min\{r_1(s_1), r_2(s_2)\}} \right\} \leq \min_{(t_1,t_2) \prec^L_1 (s_1,s_2)} \left\{ \frac{1 - \displaystyle\sum_{(t'_1,t'_2) \succ^L_1 (t_1,t_2)} \boldsymbol{P}_1(s_1,t'_1)\boldsymbol{P}_2(s_2,t'_2)}{1 - \displaystyle\sum_{(t'_1,t'_2) \succ^L_1 (t_1,t_2)} \boldsymbol{P}'_1(s_1,t'_1)\boldsymbol{P}'_2(s_2,t'_2)} \right\}$$

Let (t_1, t_2) be the state for which the ratio on the right hand side is at a minimum. Since $t_1 \prec_1 s_1$ by definition of $\prec^L_1$, we know that the following relation holds:

$$\max\left\{ B^1_{comp}, \frac{r'_1(s_1)}{r_1(s_1)} \right\} \leq \frac{1 - \displaystyle\sum_{t'_1 \succ_1 t_1} \boldsymbol{P}_1(s_1, t'_1)}{1 - \displaystyle\sum_{t'_1 \succ_1 t_1} \boldsymbol{P}'_1(s_1, t'_1)}$$

Furthermore, since by definition of the comparative bounds, $B^1_{comp} \geq \frac{r'_2(s_2)}{r_2(s_2)}$, and $B^3_{comp} \leq \min\{B^1_{comp}, B^2_{comp}\}$, we can infer that:

$$\max\left\{ B^3_{comp}, \frac{r'_1(s_1)}{r_1(s_1)}, \frac{r'_2(s_2)}{r_2(s_2)} \right\} \leq \max\left\{ B^1_{comp}, \frac{r'_1(s_1)}{r_1(s_1)} \right\}$$

To complete the proof, we make use of Observation D.3 from the proof of Theorem 6.4.12 (Montonicity), and the fact that t_2 must be the smallest state in $(S_2, \prec_2)$

by the definition of $\prec_1^L$:

$$
\begin{aligned}
\max\left\{B^3_{comp}, \frac{\min\{r_1'(s_1), r_2'(s_2)\}}{\min\{r_1(s_1), r_2(s_2)\}}\right\} &\leq \max\left\{B^3_{comp}, \frac{r_1'(s_1)}{r_1(s_1)}, \frac{r_2'(s_2)}{r_2(s_2)}\right\} \\
&\leq \max\left\{B^1_{comp}, \frac{r_1'(s_1)}{r_1(s_1)}\right\} \\
&\leq \frac{1 - \sum\limits_{t_1' \succ_1 t_1} \boldsymbol{P}_1(s_1, t_1')}{1 - \sum\limits_{t_1' \succ_1 t_1} \boldsymbol{P}_1'(s_1, t_1')} \\
&= \frac{1 - \sum\limits_{t_1' \succ_1 t_1} \boldsymbol{P}_1(s_1, t_1') \sum\limits_{t_2' \in S_2} \boldsymbol{P}_2(s_2, t_2')}{1 - \sum\limits_{t_1' \succ_1 t_1} \boldsymbol{P}_1'(s_1, t_1') \sum\limits_{t_2' \in S_2} \boldsymbol{P}_2'(s_2, t_2')} \\
&\leq \frac{1 - \sum\limits_{(t_1', t_2') \succ_1^L (t_1, t_2)} \boldsymbol{P}_1(s_1, t_1') \boldsymbol{P}_2(s_2, t_2')}{1 - \sum\limits_{(t_1', t_2') \succ_1^L (t_1, t_2)} \boldsymbol{P}_1'(s_1, t_1') \boldsymbol{P}_2'(s_2, t_2')} \\
&= \min_{(t_1, t_2) \prec (s_1, s_2)} \left\{\frac{1 - \sum\limits_{(t_1', t_2') \succ_1^L (t_1, t_2)} \boldsymbol{P}_1(s_1, t_1') \boldsymbol{P}_2(s_2, t_2')}{1 - \sum\limits_{(t_1', t_2') \succ_1^L (t_1, t_2)} \boldsymbol{P}_1'(s_1, t_1') \boldsymbol{P}_2'(s_2, t_2')}\right\}
\end{aligned}
$$

The proof of stochastic ordering under $(S_1 \times S_2, \prec_2^L)$ follows similarly.

Thus the context-bounded rate-wise stochastic ordering is preserved by $\circledast$. ■

Theorem 6.4.15. *If* $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I}) \leq_{\mathrm{rst}} \boldsymbol{Q}'_a = r'_a(\boldsymbol{P}'_a - \boldsymbol{I})$ *and for all* $s \in S$, $r_a(s) \leq r'_a(s)$, *and if* $\boldsymbol{Q}_b = r_b(\boldsymbol{P}_b - \boldsymbol{I}) \leq_{\mathrm{rst}} \boldsymbol{Q}'_b = r'_b(\boldsymbol{P}'_b - \boldsymbol{I})$ *and for all* $s \in S$, $r_b(s) \leq r'_b(s)$, *then* $\boldsymbol{Q}_a + \boldsymbol{Q}_b \leq_{\mathrm{st}} \boldsymbol{Q}'_a + \boldsymbol{Q}'_b$

Proof: Theorem 6.4.5 states that $\boldsymbol{Q}_a \leq_{\mathrm{rst}} \boldsymbol{Q}'_a$ implies that $\boldsymbol{Q}_a \leq_{\mathrm{st}} \boldsymbol{Q}'_a$, and similarly for $\boldsymbol{Q}_b$ and $\boldsymbol{Q}'_b$. If we uniformise these matrices, we have:

$$\begin{aligned} \boldsymbol{Q}_a &= \lambda_a(\overline{\boldsymbol{P}_a} - \boldsymbol{I}) \\ \boldsymbol{Q}'_a &= \lambda_a(\overline{\boldsymbol{P}'_a} - \boldsymbol{I}) \\ \boldsymbol{Q}_b &= \lambda_b(\overline{\boldsymbol{P}_b} - \boldsymbol{I}) \\ \boldsymbol{Q}'_b &= \lambda_b(\overline{\boldsymbol{P}'_b} - \boldsymbol{I}) \end{aligned}$$

where λ_a and λ_b are uniformisation constants. It follows that $\overline{\boldsymbol{P}_a} \leq_{\mathrm{st}} \overline{\boldsymbol{P}'_a}$ and $\overline{\boldsymbol{P}_b} \leq_{\mathrm{st}} \overline{\boldsymbol{P}'_b}$, hence for all states s:

$$\begin{aligned} \sum_{s<s'} \overline{\boldsymbol{P}_a}(s,s') &\leq \sum_{s<s'} \overline{\boldsymbol{P}'_a}(s,s') \\ \sum_{s<s'} \overline{\boldsymbol{P}_b}(s,s') &\leq \sum_{s<s'} \overline{\boldsymbol{P}'_b}(s,s') \end{aligned}$$

Hence:

$$\sum_{s<s'} \overline{\boldsymbol{P}_a}(s,s') + \sum_{s<s'} \overline{\boldsymbol{P}_b}(s,s') \leq \sum_{s<s'} \overline{\boldsymbol{P}'_a}(s,s') + \sum_{s<s'} \overline{\boldsymbol{P}'_b}(s,s')$$

Therefore $\overline{\boldsymbol{P}_a} + \overline{\boldsymbol{P}_b} \leq_{\mathrm{st}} \overline{\boldsymbol{P}'_a} + \overline{\boldsymbol{P}'_b}$, which implies that:

$$\boldsymbol{Q}_a + \boldsymbol{Q}_b \leq_{\mathrm{st}} \boldsymbol{Q}'_a + \boldsymbol{Q}'_b = (\lambda_a + \lambda_b)\left(\frac{\lambda_a}{\lambda_a + \lambda_b}\overline{\boldsymbol{P}'_a} + \frac{\lambda_b}{\lambda_a + \lambda_b}\overline{\boldsymbol{P}'_b} - \boldsymbol{I}\right)$$

■

Theorem 6.4.16. *If $\boldsymbol{Q}_a = r_a(\boldsymbol{P}_a - \boldsymbol{I})$ is rate-wise monotone, and for all $s \prec s' \in S$, $r_a(s) \leq r_a(s')$, and if $\boldsymbol{Q}_b = r_b(\boldsymbol{P}_b - \boldsymbol{I})$ is rate-wise monotone, and for all $s \prec s' \in S$, $r_b(s) \leq r_b(s')$, then $\boldsymbol{Q}_a + \boldsymbol{Q}_b$ is monotone.*

Proof: Theorem 6.4.6 states that the rate-wise monotonicity of $\boldsymbol{Q}_a$ implies that $\boldsymbol{Q}_a$ is monotone, and similarly for $\boldsymbol{Q}_b$. If we uniformise these matrices, we have:

$$\begin{aligned} \boldsymbol{Q}_a &= \lambda_a(\overline{\boldsymbol{P}_a} - \boldsymbol{I}) \\ \boldsymbol{Q}_b &= \lambda_b(\overline{\boldsymbol{P}_b} - \boldsymbol{I}) \end{aligned}$$

where λ_a and λ_b are uniformisation constants. It follows that $\overline{\boldsymbol{P}_a}$ and $\overline{\boldsymbol{P}_b}$ are both monotone, hence for all states s and s' such that $s \prec s'$:

$$\begin{aligned} \sum_{s' \prec s''} \overline{\boldsymbol{P}_a}(s', s'') &\leq \sum_{s \prec s''} \overline{\boldsymbol{P}_a}(s, s'') \\ \sum_{s' \prec s''} \overline{\boldsymbol{P}_b}(s', s'') &\leq \sum_{s \prec s''} \overline{\boldsymbol{P}_b}(s, s'') \end{aligned}$$

Hence:

$$\sum_{s' \prec s''} \overline{\boldsymbol{P}_a}(s', s'') + \sum_{s' \prec s''} \overline{\boldsymbol{P}_b}(s', s'') \leq \sum_{s \prec s''} \overline{\boldsymbol{P}_a}(s, s'') + \sum_{s \prec s''} \overline{\boldsymbol{P}_b}(s, s'')$$

Therefore $\overline{\boldsymbol{P}_a} + \overline{\boldsymbol{P}_b}$ is monotone, which implies that:

$$\boldsymbol{Q}_a + \boldsymbol{Q}_b = (\lambda_a + \lambda_b)\left(\frac{\lambda_a}{\lambda_a + \lambda_b}\overline{\boldsymbol{P}_a} + \frac{\lambda_b}{\lambda_a + \lambda_b}\overline{\boldsymbol{P}_b} - \boldsymbol{I}\right)$$

is also monotone. ■