University of London
Imperial College London
Department of Computing

Performance Trees: A Query Specification Formalism For Quantitative Performance Analysis

Tamas Suto

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, August 2008

Abstract

Real-life systems are often plagued by unanticipated performance problems caused by subtle bugs and bottlenecks. It is thus essential for system designers and engineers to have an understanding of their fundamental performance characteristics, both before and after implementation. Stochastic modelling and analysis respectively provide the means to abstract systems as mathematical descriptions and to derive quantifiable measures of interest from them.

A major, and so far largely unaddressed, challenge is the specification of complex performance queries on models in an accessible manner that does not sacrifice expressiveness. This thesis attempts to address this challenge by introducing Performance Trees, a new formalism for the graphical specification of complex performance queries on stochastic models. Performance Trees are designed to be accessible by providing a more intuitive approach to query specification, expressive by being able to reason about a far broader range of concepts than current alternatives, extensible by supporting additional user-defined concepts, and versatile through their applicability to multiple modelling formalisms. Performance Trees are presented in the context of a rigorous formal framework that defines the syntax, typing and quantitative semantics of operators.

Prototype tool support is implemented in the form of a module of the *PIPE2* Petri net tool, which provides graphical user interfacing and Performance Tree query design capabilities. Query evaluation is supported by a set of integrated parallel and distributed analysis tools, and realised by the distribution of computations onto a dedicated Grid cluster.

The practical application of Performance Trees is demonstrated in the context of case study analysis scenarios of an electronic voting system, an online transaction system and a hospital's Accident & Emergency unit.

Acknowledgements

I would like to express my deepest appreciation and gratitude to:

Dr William J. Knottenbelt, my supervisor, for his conscientious supervision, continuous support, motivation and enthusiasm, and for making me have the best time of my life as a Ph.D. student;

Dr Jeremy T. Bradley, my co-supervisor, for his generous professional guidance, support and mentoring;

The AESOP research group, for providing such a pleasant and intellectually stimulating social environment;

My friends and colleagues, for their support and for making this Ph.D. such a truly enjoyable, enriching and memorable experience;

Dalal Alrajeh, Douglas de Jager, Dorian Gaertner, Richard Hayden, Jaspreet Shaheed, Dr David Thornley, Maria Grazia Vigliotti and Silvana Zappacosta, for being a constant source of delightful and refreshing entertainment;

David McBride, Nick Dingle, Jay Jayasundera, Matt Johnson, Stuart McRobert, Simon Tagg and Duncan White, for their kind help and expert technical advice throughout my time at the Department of Computing;

The Department of Computing at Imperial College London, for providing first class facilities and a well-equipped, comfortable and productive working environment;

The Engineering and Physical Sciences Research Council, for generously supporting my research;

My family, for their never-ending support, encouragement and love, and for giving me the opportunity to fulfill my dreams.

Dedication

I wish to dedicate this thesis to my parents, who are the best that a child could ever hope for, and whom I love above all.

“It is an immutable law in business that words are words, explanations are explanations, promises are promises, but only performance is reality.”

Harold S. Geneen, CEO of ITT Corporation (1959-1972)

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the last few decades, we have witnessed scientific and technological advancement on an unprecedented scale. This has had an enormous impact on humanity. We find ourselves surrounded by complex computer systems supporting and even governing many aspects of our lives. Due to this strong dependence on technology, it is a basic necessity that systems function correctly and reliably, while also exhibiting good performance. However, due to the inherent complexity of today's systems, the prediction of their performance is often difficult. Performance-related analysis provides a rigorous way to address this problem by examining, among others, critical system characteristics, such as availability, reliability, responsiveness and efficiency.

A system is generally considered to be available if it has not failed and is not undergoing maintenance. Many systems are required to be available as often and as consistently as possible in order to serve customers who expect and rely on continuous availability. Web servers and eCommerce infrastructures are good examples of such systems. It is imperative that measures of availability be accessible to system engineers to enable them to anticipate and eliminate undesired service interruptions.

Reliability, the probability of a system performing a specified function without failure under given conditions for a specified period of time, is particularly crucial to systems that either support mission-critical applications or are exposed to extremely high user demand. Nuclear power plant and airspace control systems are classic examples of the former, and local area and telephone networks of the latter kind of systems. It is generally considered unacceptable for such systems to fail unexpectedly. Reliability-oriented analyses are essential to gaining an insight into the likelihood of system failure.

Responsiveness is an indicator of the speed with which systems respond to requests. Users invariably want to be able to interact with highly responsive systems in order to maximise their perceived productivity. Often, a more responsive system can have a greater effect on overall user satisfaction than the fast completion of an associated request. Therefore, good system design also takes responsiveness criteria into account and benefits from responsiveness-oriented system analysis [Powell02].

Efficiency, an indicator of the actual performance of systems compared to their possible performance, is also an important system property for designers and engineers to consider. This is so because it is often a requirement for systems to operate at peak efficiency, especially in industry, where complex IT systems support the large majority of business operations. Businesses often find themselves in positions where they may be in a strategically more advantageous position than a competitor, simply because their systems perform more efficiently. An investment bank's electronic trading platform is a fitting example, since even a minor difference in terms of its efficiency compared to a that of another bank's systems can make an big difference in earnings. Efficiency, as a measure of performance, is also very relevant in the context of service provision, since efficient systems have higher throughput and are as a result, on average, able to serve customers more promptly.

Quality of Service (QoS) [Meyer80] can be thought of as a measure that represents a particular level of performance and characterises factors such as availability, reliability and responsiveness of systems or services. QoS-oriented performance analysis has a wide range of applications in helping service providers to guarantee certain levels of service. Telecommunications and network providers, for instance, rely heavily on performance analysis to ascertain whether or not expected levels of QoS are being provided to customers, while the health care sector employs performance analysis to monitor whether government-set QoS targets are being met consistently [Au-Yeung04].

For engineers to be able to design systems that conform to strict QoS requirements, accessible ways of modelling them and analysing their performance are necessary. Analysis is traditionally carried out by creating a model of a system, in order to mathematically abstract its behaviour in a way that is amenable to analysis, and constructing a query that defines performance properties of interest. Performance queries are then evaluated on the applicable model to obtain a quantifiable performance metric – such as that addressed by the query *"In a hospital waiting room, what is the steady-state distribution of the number of patients waiting to be treated?"* – or to determine whether the system conforms to a particular QoS requirement – as set out in the query *"In a mobile communications network, is the time taken to send an SMS message between two handsets less than 5 seconds with more than 95% probability?"*

In performance analysis, stochastic models (see Section 2.1) are needed to represent real-life systems, since they are able to take into account their intrinsic probabilistic nature and mirror their essential behaviour, while omitting details that would unnecessarily increase the complexity of evaluation. An important advantage of using models for the performance analysis of real-life systems is that if a system can be solved analytically, it is typically relatively simple to obtain performance measures from it. This also allows for a more accurate evaluation of performance aspects than obtainable from prototypes or simulations. Hence, good models can provide the best opportunity for observing trends that emerge from a system's behaviour.

After constructing a stochastic model of a system, its performance can be analysed by specifying and evaluating performance queries, of which there are mainly two kinds. Queries that aim to verify conformity to QoS constraints are called *performance requirement* queries, while queries that aim to obtain metrics that characterise model performance in some way are called *performance measure* queries. Performance requirement queries have traditionally been expressed in formulae of stochastic logics (see Section 2.2.2) and evaluated by model checkers (see Section 2.4.1), while performance measure queries have been specified in tool-specific languages (see Section 2.2.4) and evaluated by quantitative analysers.

We have identified certain aspects of the traditional performance analysis process that represent significant research opportunities and which have provided the motivation for the present research:

1. There is a strict separation between the specification of performance requirement and performance measure queries. At present, no unified environment exists for their common specification, and certainly none for their common evaluation. Hence, queries consisting of both performance measure- *and* requirement-oriented concepts cannot be specified or evaluated.

2. Current performance query specification formalisms lack accessibility. That is, they require specialist knowledge and expertise to be used effectively, which is often not compatible with the background knowledge of typical system designers and engineers. This is the case with logical formalisms and tool-specific languages, for example, as also noted by [Grunske08a]. Furthermore, prevalent logical paradigms

may also seem esoteric to many industrial users, and diverse tool-specific specification languages have little in common with one another. This may be one of the reasons why many industrial users rather resort to simulations for the purposes of performance analysis.

3. The scope of expressiveness of current performance query specification formalisms is overly constrained. This is because such formalisms only support the specification of a few, relatively basic, performance properties, and do not provide the means to reason about more advanced concepts (such as distributions, densities, convolutions, moments and percentiles, for example). Logical representations are concise and rigorous, but they are specific to performance verification, and hence not applicable in a wide range of scenarios. Tool-specific languages may be able to address performance measures, but they are limited in terms of expressiveness by the tools that implement and support them.

1.2 Objectives

The research presented in this thesis has two main objectives:

1. To develop a novel performance query specification formalism that
   - is accessible to system designers and engineers by providing a simple and intuitive approach to query construction;
   - enables the expression of performance queries utilising concepts related to both performance requirement specification *and* quantitative measure extraction;
   - expands on the expressiveness of current query specification formalisms by supporting a wider range of performance concepts than presently possible.
2. To implement an integrated performance analysis environment that provides tool support for the accessible design of stochastic system models and performance queries (using the newly developed specification formalism), and is able to evaluate them in a large-scale parallel and distributed fashion on a Grid-based computational back-end that harnesses the power of a range of dedicated performance analysis tools.

1.3 Contributions

1.3.1 The Performance Tree Formalism

This thesis introduces *Performance Trees*, a novel graphical performance query specification formalism that enables the expression of complex queries containing both performability requirements and quantitative measures. Performance Trees support a wide range of concepts, applicable to stochastic system models, that are likely to be familiar to system designers and performance engineers. They represent an alternative to traditional approaches to query specification, which have so far mostly been based on complex logical and textual formalisms. Performance Trees ease the process significantly by providing a convenient and accessible way of specifying performance queries with their visual hierarchical tree structure that allows performance queries to be composed graphically.

Their abstract state specification mechanism equips Performance Trees with a certain degree of versatility that other formalisms often do not have, by allowing queries to be defined over a number of different modelling formalisms. This thesis presents Performance Trees in the context of generalised stochastic Petri nets (GSPNs) and the stochastic process algebra PEPA. The formalism is also capable of extracting customer-centric performance measures from such models, which allow queries to be specified that involve reasoning about individual customers. In addition, Performance Trees do not place any artificial constraints on the size of models that can be solved, since they are evaluated by a set of analysis tools, whose inherent solution capacity determines the scope of evaluation.

Performance Trees are an extensible formalism. Through the use of parameterised macros, custom performance concepts can be incorporated into the set of available operators and reused in other queries. Such user-defined macros are constructed from the set of basic Performance Tree operators. New operators, representing additional performance concepts distinct from the ones that are already available, can be incorporated into the formalism, provided that evaluation support in the form of analysis tools is also integrated into the underlying analysis framework.

1.3.2 Formal Characterisation of Performance Trees

We also present a formal characterisation, which includes the syntax, typing and quantitative semantics of Performance Tree operators.

1.3.3 Tool Support for Performance Trees

Tool support for the graphical specification of Performance Tree queries is realised by an enhanced version of the open-source Petri net editor *PIPE2*, which enables the graphical specification of GSPN models. We have implemented the Performance Query Editor module to provide an interactive graphical interface that allows users to design Performance Tree queries on system models defined in *PIPE2*, submit these for evaluation, track evaluation progress, and visualise obtained results. As such, the module also serves as the client front-end to a sophisticated performance analysis environment that enables the parallel and distributed evaluation of Performance Tree queries on GSPN models. With the Performance Query Editor module, *PIPE2* has been extended to provide a single point of access to Performance Tree-based query specification and evaluation.

The analysis environment's evaluation back-end consists of a server that communicates with *PIPE2* to coordinate the evaluation of queries, and a Grid-based computational resource pool that integrates high performance hardware and a wide range of specialised performance analysis tools. The evaluation back-end uses caching to ensure that queries that have been evaluated already are not processed again when evaluation is requested on the same model. Evaluation results are stored on disk and retrieved in case of repeated requests. Integrated smart scheduling ensures that, where possible, Performance Tree queries are evaluated concurrently.

1.3.4 Application of Performance Trees in Case Study Scenarios

This thesis also presents the practical application of Performance Trees in GSPN-based case study performance evaluation scenarios, including an electronic voting system, an online transaction system and a hospital's Accident & Emergency unit.

1.4 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 presents relevant background material. An overview of stochastic modelling is given, describing low- and high-level formalisms for the abstract representation of real-life systems. Traditional approaches to performance query specification are also discussed by providing an overview of performance query classification and considering logical, graphical and tool-specific languages. Classical methods of performance analysis are then considered, with particular emphasis on probabilistic model checking, numerical analysis methods and simulation. Finally, a summary of the most relevant currently available tools for performance analysis is provided.

Chapter 3 introduces Performance Trees, a novel formalism for the representation of performance queries. An introduction to the formalism, together with a description of its structure and the set of available operators is given. Following that, the power of Performance Trees is highlighted by discussing their accessibility, expressiveness, extensibility and versatility, and demonstrating their practical application on a number of example performance analysis scenarios.

Chapter 4 details the formal characterisation of Performance Trees. We present the syntax, typing and quantitative semantics of Performance Tree operators, which together form the formalism's theoretical framework.

Chapter 5 discusses the modelling and analysis of real-life systems in an integrated parallel and distributed performance analysis environment. This environment consists of a number of interacting software and hardware components. *PIPE2*, a Java-based open-source Petri net design tool, serves as the user-facing graphical interface and provides functionality for the specification of GSPN system models and Performance Tree queries. It interacts with the Analysis Server, the coordinating component of the analysis environment. The Analysis Server optimises query evaluations and outsources computations to a number of specialised parallel and distributed tools, which carry out computations on a dedicated analysis cluster. Following the description of the analysis environment's architecture, the analysis process is discussed in detail.

Chapter 6 presents case study evaluation scenarios to demonstrate the application of Performance Trees. Analyses of an electronic voting system, an online transaction system and a hospital's Accident & Emergency unit are described.

Chapter 7 concludes the thesis by summarising and evaluating the theoretical and practical contributions, discussing areas of application of the research and highlighting opportunities for future work.

Appendix A provides detailed descriptions of the models used in the case study evaluations.

1.5 Publications

The publications summarised below have arisen as part of the research carried out during the course of this Ph.D.

1. **Workshop on Process Algebra and Stochastically Timed Activities** (PASTA'05) [Suto05] describes early efforts aimed at finding an accessible and user-friendly approach to the specification of QoS-related performance requirements. The idea presented in this paper considers the development of an extended form of stochastic logic, to serve as the underlying theoretical framework for a user-facing graphical performance query specification front-end. This graphical front-end intends to provide a layer for visual query composition and to hide from the user the complexities involved in the specification of logical performance queries. Queries defined in the graphical formalism are proposed to be translated into the stochastic logic for evaluation purposes. The paper also introduces the idea of utilising the greatly extended computational power that can be provided by a Grid infrastructure for performance analysis by distributing, parallelising and optimising applicable model-checking computations on a dedicated Grid-based analysis cluster.

2. **Workshop on Process Algebra and Stochastically Timed Activities** (PASTA'06) [Suto06a] departs from [Suto05]'s original idea of using an extended stochastic logic for performance requirement representation, but retains and expands on the concept of graphical performance specification. The paper introduces Performance Trees, a novel formalism for the graphical specification of performance queries on stochastic models. Performance Trees represent performance queries as visualised hierarchical tree structures, and aim to provide an accessible alternative to stochastic logics, the thus far prevalent means of performance requirement specification. The range of Performance Tree operators allows the expression of performance requirement-oriented queries and also provides the ability to specify quantitative measures of interest on stochastic models. Material from this paper appears in Chapter 3.

3. **International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems** (MASCOTS'06) [Suto06b] expands on work presented in [Suto06a] by investigating and classifying the different kinds of performance queries that may be of relevance to system designers and engineers, and describing how Performance Trees can be used as a query specification formalism that is able to express performance requirements and performance measures at

the same time. The paper introduces the ability of Performance Trees to reason about passage time distributions and densities, transient and steady state measures and moments. It also presents the syntax and type system for Performance Trees and provides an outline mapping from concepts that can be expressed by the stochastic logic CSL to Performance Tree operators. The paper also illustrates how semi-Markov passage time computation algorithms, based on numerical Laplace transform inversion, can be directly applied to the resolution of a case study Performance Tree query on a GSPN model of a voting system. Material from this paper is used in Chapters 3, 4 and 6.

4. **International Conference on the Quantitative Evaluation of Systems** (QEST'07) [Suto07] expands on work presented in [Suto06b] by solidifying the theoretical foundations of Performance Trees through the provision of quantitative semantics, which define the mathematics underlying individual Performance Tree operators. The paper also focuses on illustrating differences in terms of expressiveness between Performance Trees and CSL, and presents a case study performance query specification on a GSPN model of a hospital's Accident & Emergency unit. Material from this paper forms part of Chapter 4.

5. **International Workshop on Parallel and Distributed Methods in Verification** (PDMC'08) [Brien08b] describes the first realisation of an evaluation environment for Performance Trees. In particular, the paper presents details of the architecture and implementation of this environment, comprising of a client-side model and performance query specification tool, a server-side distributed evaluation engine, and a dedicated Grid cluster. The evaluation engine combines the analytic capabilities of a number of distributed tools for steady state, passage time and transient analysis, and also incorporates a caching mechanism to avoid redundant calculations. The paper describes the analysis process and demonstrates in the context of a case study of a hospital's Accident & Emergency unit how this analysis environment allows remote users to design models and performance queries in a sophisticated, yet easy-to-use framework, and subsequently evaluate them by harnessing the computational power of a Grid back-end. Material from this paper can be found in Chapter 5.

6. **SPEC International Performance Evaluation Workshop** (SIPEW'08) [Bradley08] considers recent developments in the analysis of stochastic process algebra models, which allow for transient measures of very large models to be extracted. By performing so-called fluid analysis of stochastic process algebra models, it is now feasible to analyse systems of $O(10^{1000})$ states and beyond. This paper extends the type of measure that can be extracted with fluid analysis, and presents a systematic

transformation of a PEPA model that enables the extraction of measures analogous to response times. It also presents a case study, which shows how response time measures can be extracted from a PEPA model of a health care system. Material from this paper is presented in Chapter 3.

7. **International Conference on the Quantitative Evaluation of Systems** (QEST'08) [Dingle08b], a tool paper, builds on [Brien08b] and presents the evaluation of Performance Tree queries on stochastic models in the context of an integrated parallel and distributed analysis environment. The graphical user interface to this environment is implemented in the *PIPE2* tool, a Java-based open-source Petri net editor, which provides query design capabilities and control over query evaluation. Evaluation is coordinated by the Analysis Server, which is responsible for the scheduling of jobs on a Grid-based computational cluster that integrates a number of specialised parallel and distributed analysis tools. Material from this paper appears in Chapter 5.

8. **IEEE Transactions on Software Engineering** (submitted for publication) [Suto08a] builds on material presented in [Dingle08b] and addresses the integration of Performance Trees into a parallel and distributed performance analysis environment. The paper describes how support for tagged customers in system models and the application of Performance Trees in their context is realised. It also reasons about how Performance Trees can be used to query system models based on the stochastic process algebra PEPA, and presents a case study performance evaluation of a hospital's Accident & Emergency unit – demonstrating some of the capabilities of the analysis environment. Material from this paper is incorporated into Chapters 5 and 6.

9. **ACM SIGMETRICS Performance Evaluation Review (Special Issue)** (invited paper) [Suto08b] builds on [Brien08b] and [Dingle08b], and discusses *PIPE2*, an open-source tool for GSPN-based system modelling and analysis. *PIPE2* was originally developed as a platform-independent Petri net editor, to support the design of complex GSPN-based system models. Subsequently, it has been enhanced with a number of analysis modules and has evolved into a versatile front-end for a sophisticated parallel and distributed performance evaluation environment. With *PIPE2*, users are able to design and evaluate complex performance queries – expressed in the Performance Tree formalism – primarily aimed at performance property verification and performance measure extraction. The paper provides an overview of *PIPE2*'s features and discusses details of its underlying evaluation environment. Material from this paper is detailed in Chapters 5 and 6.

10. **International Conference on Parallel, Distributed and Grid Computing for Engineering** (PARENG'09, invited book chapter) [Knottenbelt09] describes how Performance Trees attempt to address the challenge of specifying complex performance queries on models of systems in a way that is both accessible and expressive. It elaborates on their ability to provide more intuitive query specification, to reason about a broader range of concepts than current alternatives, to support additional user-defined concepts, and to express queries on multiple underlying modelling formalisms. The paper describes in detail tool support for Performance Trees, and presents a natural language-based query builder interface for *PIPE2*. Details on parallel and distributed query evaluation of Performance Tree queries are given, and their application is demonstrated in the context of a case study of an online transaction system. The flexibility of the formalism is further illustrated by extensions that permit the specification and monitoring of Service Level Agreements. Material from this paper has been used in Chapters 3 to 6 and Appendix A.

1.6 Statement of Originality

I declare that this thesis was composed by myself, and that the work it presents is my own, except where otherwise stated.

Chapter 2

Background

This chapter provides an overview of background material that is relevant to the research presented in this thesis. Specifically, it discusses stochastic modelling, methods and tools for performance analysis, traditional approaches to performance query specification and performance evaluation in Grid environments.

2.1 Stochastic Modelling

It is generally of great interest to system designers and engineers to have an understanding of the way in which systems behave. There are two approaches that are primarily used for the investigation of factors that have an effect on critical system properties: experimentation and analysis. Experimentation is generally considered expensive, time-consuming and unreliable, due to a lack of coverage of all possible scenarios. It can, however, be useful for thorough explorations of the effect that specific parameters have on system behaviour. Analysis, in contrast, is cheap, effective and dependable. In order to enable tractable analyses of systems, mathematical models that abstract system behaviour need to be constructed. These are quantitative system descriptions that approximate reality by making simplifying assumptions and omitting non-essential details. If a model can be solved analytically, it is typically relatively simple to obtain performance measures from it. Because of this, good models provide the best means for the discovery of trends that can provide an understanding of qualitative aspects of system performance. However, the necessary restrictions on the level of detail embedded in models often result in inaccurate representations of absolute performance. Nevertheless, relative performance is usually a more than satisfactory measure, since it can be used to investigate different configurations of the same system [Mitrani98].

Mathematical models come in different flavours. *Deterministic models* predict a single outcome from a given set of possible outcomes, while *stochastic models* predict a set of possible outcomes, weighted by their likelihoods. Since real-life systems exhibit randomness, modelling tools that are needed to study them come from the domains of probability theory. Stochastic models are used widely in many areas of the natural and engineering sciences, since they can represent the behaviour of both natural and man-made systems.

2.1.1 Markov and Semi-Markov Processes

Random Variables

Most stochastic models are expressed in terms of random variables. A random variable represents the outcome of a random experiment and is therefore characterised by its probabilistic outcome. Random variables can represent, for instance, the number of customers in a system, the time a customer takes to traverse a system, the proportion of time that there are fewer than k customers in a system, etc. A random variable is *discrete* if it can only have a finite number of values, and *continuous* otherwise.

Definition A *random variable* χ on a sample space $\mathcal{S}$ is a function $\chi : \mathcal{S} \to \mathbb{R}$ that assigns a real number $\chi(s)$ to each random outcome $s \in \mathcal{S}$ [Nelson95].

Values of random variables can be specified probabilistically using distribution functions. For a discrete random variable, χ, the *probability mass function (pmf)*, $f_\chi(x)$, gives the probability of the random variable being equal to some value, x:

$$f_\chi(x) = \mathbb{P}(\chi = x) = \sum_{\chi(s)=x} \mathbb{P}(s) \tag{2.1}$$

For continuous random variables, the *probability density function (pdf)*, $f(x)$, is the derivative of a random variable's probability distribution, $F(x)$:

$$f(x) = \frac{d}{dx}F(x); \tag{2.2}$$

Informally, a pdf can be thought of as a smoothed-out version of a histogram. If enough values of a continuous random variable are sampled, producing a histogram depicting relative frequencies of output ranges, then this histogram will resemble the random variable's probability density, assuming that the output ranges are sufficiently narrow. Values of a pdf are not probabilities themselves, but instead, the integral of a pdf over a range of possible values $(a, b]$ gives the probability of the random variable falling within that range.

The probability that a random variable, $\mathcal{X}$, takes on a value that does not exceed a given number, x, is given by the *cumulative distribution function (cdf)*, $F(x)$:

$$F(x) = \mathbb{P}(\mathcal{X} \leq x) = \begin{cases} \sum_{t \leq x} f_{\mathcal{X}}(t), & \text{if } \mathcal{X} \text{ is discrete} \\ \\ \int_{-\infty}^{x} f(t)\, dt, & \text{if } \mathcal{X} \text{ is continuous.} \end{cases} \tag{2.3}$$

The pdf can also be used for the expression of the probability of a random variable, $\mathcal{X}$, taking on a value in the interval $(a, b]$:

$$\mathbb{P}(a < \mathcal{X} \leq b) \;=\; \int_{a}^{b} f(x)\, dx \tag{2.4}$$

A random variable is characterised completely by its probability distribution or its probability density function. However, it is sometimes desirable, and sufficient for practical purposes, to describe a random variable by its summary statistics [Trivedi02].

One important measure of a random variable, $\mathcal{X}$, is its *expectation*, $E(\mathcal{X})$, which is formally defined as:

$$E(\mathcal{X}) \;=\; \begin{cases} \sum_{i} x_i p(x_i), & \text{if } \mathcal{X} \text{ is discrete} \\ \\ \int_{-\infty}^{\infty} x f(x)\, dx, & \text{if } \mathcal{X} \text{ is continuous.} \end{cases} \tag{2.5}$$

Exponential distributions are a very important type of probability distribution, which arise naturally when modelling the time between independent events that happen at a constant average rate. A random variable, $\mathcal{X}$, taking on non-negative real values, is said to be exponentially distributed if its cdf has the form:

$$F(x) \;=\; 1 - e^{-\lambda x}; \quad x \geq 0 \tag{2.6}$$

This function depends on a single parameter, $\lambda > 0$, which is called the *rate*. The corresponding pdf is:

$$f(x) = \lambda e^{-\lambda x}; \quad x \geq 0 \tag{2.7}$$

Exponentially distributed random variables are commonly used to model random time intervals with arbitrary lengths. $\mathcal{X}$ may represent the service of a job, the duration of a communication session, the interval between consecutive arrivals, etc. An important property of the exponential distribution is that its future progress does not depend on its past. This is also known as the *memoryless property*. Another way of expressing this property is to say that the probability of an activity continuing for another interval of length at least y, given that it has already lasted for time x, is independent of x. The following equation [Nelson95] clarifies the reason for this:

$$\mathbb{P}(\mathcal{X} > x + y \mid \mathcal{X} > x) = \frac{1 - F(x+y)}{1 - F(x)} = e^{-\lambda y} = \mathbb{P}(\mathcal{X} > y) \tag{2.8}$$

Stochastic Processes

Definition A *stochastic process* is a set of random variables $\{\mathcal{X}(t) : t \in \mathcal{T}\}$, indexed by the time parameter t.

Typically, $\mathcal{T}$ represents a set of points in time, and $\mathcal{X}(t)$ the value of the stochastic process at time t, which is also referred to as its *state*. The state space of the process is the set of all possible values that $\mathcal{X}(t)$ can assume. Stochastic processes are classified according to time, and we say that they are *discrete*- or *continuous-time*, depending on whether $\mathcal{T}$ is discrete or continuous. For continuous-time stochastic processes, $\mathcal{T} = \mathbb{R}^+$. Stochastic processes offer a way to capture complex forms of dependency between sets of random variables, which is why stochastic models make use of them. If we take the random variable $\mathcal{X}(t)$ to be the total number of customers that have arrived at a system over the time period $[0, t]$, $\mathcal{X}(t)$ is called a *counting process*. A special type of such a process is a renewal process.

Definition Let $S_1, S_2, S_3, \ldots$ be a sequence of independent, identically distributed random variables, such that $0 < E(S_i) < \infty$. We refer to the random variable S_i as the i^{th} *holding time*. We define the n^{th} *jump time* as

$$J_n = \sum_{i=1}^{n} S_i. \tag{2.9}$$

The intervals $[J_n, J_{n+1}]$ are called *renewal intervals*. Then, the random variable χ_t given by

$$\chi_t = \max\{n : J_n \leq t\}, \quad t \geq 0 \tag{2.10}$$

is called a *renewal process* [Nelson95]. In the context of real-life systems, we may think of the holding times $\{S_i : i \geq 1\}$ as the elapsed time before a system breaks for the i^{th} time since the last breakdown. In this context, the jump times $\{J_n : n \geq 1\}$ record the successive times at which the system breaks, and the renewal process χ_t records the number of times the system has broken down by time t. Renewal processes are often found embedded in other stochastic processes, most notably Markov processes [Pyke61a].

When attempting to build a realistic stochastic model of a physical scenario, dependencies need to be taken into account. For example, purchases made at the supermarket next week may depend on the satisfaction with purchases made up until now; or a shop's inventory on a particular day depends on the stock level on the previous day, as well as on customer demand; or the number of customers awaiting service at a facility depends on the number of waiting customers in previous time periods. Dependencies ensure realistic models, but at the same time make probability calculations very difficult or even impossible.

The more independence is exhibited by a model, the greater the possibility for explicit calculations, but the more questionable the level of realism that is inherent to the model. Hence, when constructing a model, the challenge lies in maintaining dependencies that ensure sufficient realism, but which at the same time do not make mathematical tractability infeasible. In most cases, the future behaviour of a system depends to some extent on its past. That is, the states χ_{n_1} and χ_{n_2}, at two different moments in time, are dependent random variables. A particular class of stochastic process, which exhibits the Markov property, has a limited form of state dependency [Nelson95].

Definition The *Markov property* states that given the current state of a stochastic process, χ_n, the distribution of any future state, χ_f, does not depend on the past history of the process, $\chi_p : p < n$. In other words, the present state of the process contains all the information about its past that is needed to determine its future evolution [Nelson95]. Formally,

$$\mathbb{P}(\chi_{n+1} = j \mid \chi_n = i_n, \chi_{n-1} = i_{n-1}, \ldots, \chi_0 = i_0) \;=\; \mathbb{P}(\chi_{n+1} = j \mid \chi_n = i_n) \tag{2.11}$$

Processes that exhibit the Markov property, called *Markov processes*, are among the most important tools of probabilistic modelling, since they make dependencies manageable. Markov processes whose state space is discrete, are referred to as *Markov chains*. The dynamic behaviour of Markov processes is characterised by the transitions between their states and the times spent in them. Generally, these *state holding times*, also often called *sojourn times*, represent the periods where some form of processing is taking place in the systems being modelled by the Markov processes. In contrast, transitions represent events in the system. The Markov property ensures that at any point, the distribution of time until the next state change is independent of the time of the previous state change. Since the only probability distribution function that exhibits this property is the exponential distribution, we know that sojourn times are exponentially distributed in Markov processes [Mitrani98].

Discrete-Time Markov Chains

Definition A *discrete-time Markov chain (DTMC)* is one whose parameter space is discrete, i.e. a stochastic sequence, $\{\chi_n \mid n = 0, 1, 2, \dots\}$, that satisfies Equation 2.11 for $n \in \mathbb{N}$. The possible values of χ_n form a countable set $\mathcal{S}$, called the state space of the DTMC. Changes of state occur at discrete time intervals [Bause02].

The evolution of a DTMC is described by so-called *one-step transition probabilities*, p_{ij}, of the chain moving to state j at time $n+1$, given that it is in state i at time n:

$$p_{ij} = \mathbb{P}(\chi_{n+1} = j \mid \chi_n = i) \tag{2.12}$$

In the case of *time-homogeneous Markov chains*, the behaviour of a system does not depend on when it is observed, since transition probabilities between states are independent of the time at which transitions occur, and hence do not change over time [Hillston04]:

$$p_{ij} = \mathbb{P}(\chi_{n+1} = j \mid \chi_n = i) = \mathbb{P}(\chi_{n+m+1} = j \mid \chi_{n+m} = i) = p_{ij} \tag{2.13}$$

where $n = 1, 2, \dots,\ \ m \geq 0,\ \ i, j \in \mathcal{S}$. One-step transition probabilities can be specified compactly in the form of a transition probability matrix, ***P***, which completely characterises a time-homogeneous DTMC:

$$\boldsymbol{P} = \begin{bmatrix} p_{00} & p_{01} & p_{02} & \cdots \\ p_{10} & p_{11} & p_{12} & \cdots \\ \vdots & \vdots & \vdots & \\ p_{i0} & p_{i1} & p_{i2} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} \tag{2.14}$$

Indices range over the state space, and since the chain inevitably has to be in some state at any observed instant, all rows of ***P*** sum to 1:

$$\sum_{j=0}^{\infty} p_{ij} = 1; \;\; i = 0, 1, \ldots \tag{2.15}$$

An equivalent description of the one-step transition probabilities can be given by a directed graph called the *state-transition diagram*. A node labelled i in the diagram represents state i of the DTMC, and an arc going from node i to node j, labelled p_{ij}, implies that the one-step transition probability is $p_{ij} = \mathbb{P}(\mathcal{X}_{n+1} = j \mid \mathcal{X}_n = i)$. An example state-transition diagram can be seen in Figure 2.1.

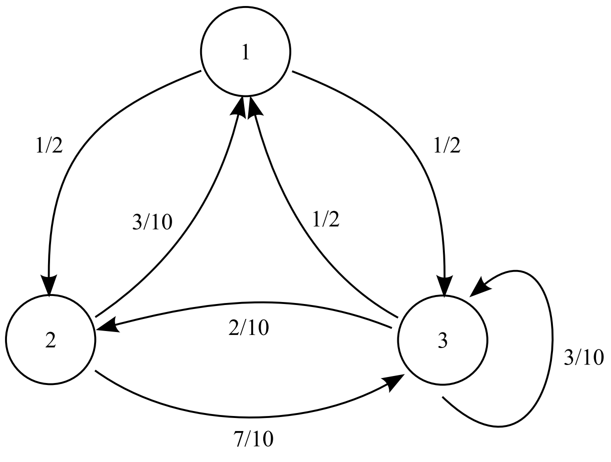


Figure 2.1: A DTMC with transition probabilities

The probability of a DTMC to be in state j, n steps after being in state i, called the *n-step transition probability*, is given by the Chapman-Kolmogorov equation:

$$p_{ij}^{n} \;=\; \mathbb{P}(\mathcal{X}_{m+n} = j \mid \mathcal{X}_m = i) \;=\; \sum_{k \in \mathcal{S}} p_{ik}^{m} p_{kj}^{n-m}; \quad 0 \le m \le n \tag{2.16}$$

To indicate that the chain can move directly from state i to state j, we write $i \mapsto j$ if $p_{ij} > 0$. The operation of the chain can be envisaged as follows. It starts at time 0 in some state $i_0 \in \mathcal{S}$. At the next time unit or step, the chain moves to a neighbouring state i_1 with probability $p_{i_0 i_1}$, provided that $i_0 \mapsto i_1$. It is possible that this move is immediately back to the state itself, which we refer to as a *self-loop*. This procedure is repeated, so that at step n, the chain is in some state i_n, where $i_0 \mapsto i_1 \mapsto \ldots \mapsto\ i_{n-1} \mapsto i_n$. A sequence of states satisfying this arrangement is called a *path*. We write $i \overset{n}{\rightsquigarrow} j$ if there exists a path of n steps between i and j, and $i \rightsquigarrow j$ if there exists a path from state i to state j [Nelson95].

The classification of states depends on the structure of the Markov chain. A state i is called *absorbing* if $i \not\rightsquigarrow j$, for any state $j \neq i$. Once entered, the system will stay in state i forever. A state i is called *transient* if starting from it, there is a positive probability that the chain will never return to it. If, for example, a state j exists, such that $i \rightsquigarrow j$ but $j \not\rightsquigarrow i$, then i is transient. A transient state can only be visited a finite number of times with probability 1. A state is called *recurrent* if the probability of eventually returning to it is 1. In such a case, clearly, $i \rightsquigarrow j$ and $j \rightsquigarrow i$ and we say that states i and j *communicate*. A Markov chain is *irreducible* if all states communicate with each other, otherwise it is *reducible*.

We call a sequence of states starting and ending at state i an i-*cycle*. If the expected number of steps in an i-cycle is finite, we call state i *positive recurrent*; otherwise state i is said to be *null recurrent*. If the i-cycle $i \overset{n}{\rightsquigarrow} i$ exists only when $n = kd$ for some values of k and a fixed value of $d > 1$, then state i is said to be *periodic* with period d. This indicates that the state can only return to itself after some multiple of d steps. States that are not periodic are called *aperiodic*. States that are positive recurrent and aperiodic are called *ergodic*. Models whose states are all ergodic, are themselves classified as ergodic. Most Markovian models that arise in applications are irreducible and ergodic [Nelson95].

Figures 2.2 and 2.3 demonstrate these concepts. State 1 in Figure 2.2 is an example of a transient state and state 3 that of an absorbing state. States 2, 4 and 5 are examples of recurrent states, and the sub-chain consisting of these states is irreducible. The chain shown in Figure 2.3 is positive recurrent and aperiodic, hence ergodic [Nelson95].

Continuous-Time Markov Chains

Continuous-time Markov chains (CTMCs) are used to model systems where changes of state can occur at arbitrary moments, and where intervals between those changes can be of arbitrary length.

Definition A *continuous-time Markov chain* is a stochastic sequence $\{\mathcal{X}(t) \mid t \geq 0\}$, with $\mathcal{X}(t) \in \mathcal{S}$, where $\mathcal{S}$ is the discrete state space of the process. By the Markov property, the

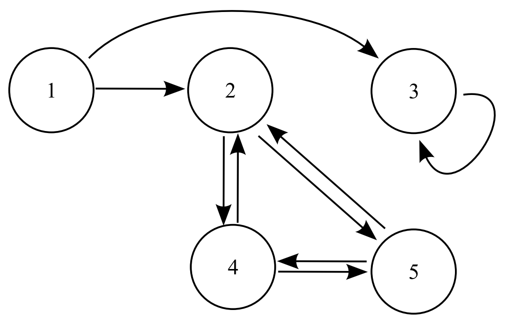


Figure 2.2: A DTMC with transient, absorbing and recurrent states

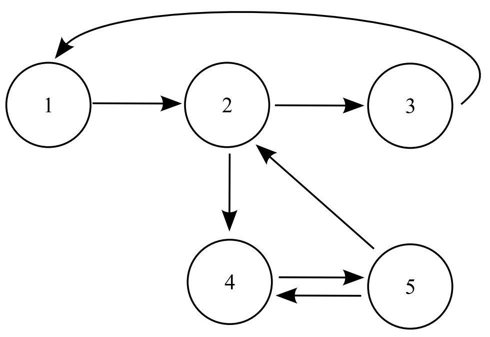


Figure 2.3: An ergodic DTMC

evolution of the CTMC after a given moment t depends only on the state at that moment, $\mathcal{X}(t)$, and not on the past behaviour [Mitrani98]:

$$\mathbb{P}(\mathcal{X}(t) = x \mid \mathcal{X}(t_n) = x_n, \ldots, \mathcal{X}(t_0) = x_0) = \mathbb{P}(\mathcal{X}(t) = x \mid \mathcal{X}(t_n) = x_n) \tag{2.17}$$

for any sequence $t_0, t_1, \ldots, t_n$, such that $t_0 < t_1 < \cdots < t_n < t$. The evolution of a CTMC is described by a *generator matrix* $\boldsymbol{Q}$, whose every element q_{ij} represents the infinitesimal rate of moving from state i to state j, where $i \neq j$:

$$\boldsymbol{Q} = \begin{bmatrix} q_{00} & q_{01} & q_{02} & \cdots \\ q_{10} & q_{11} & q_{12} & \cdots \\ \vdots & \vdots & \vdots & \\ q_{i0} & \cdots & q_{ii} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} \tag{2.18}$$

The behaviour of a typical CTMC can be described as follows. The process enters a state i, and remains in that state for a random period of time, distributed exponentially with parameter $-q_{ii}$, where $q_{ii} = -\sum\limits_{i \neq j} q_{ij}$. At the end of that period, the process moves to a different state $j \neq i$, with some probability p_{ij}. The Markov property implies that if at any moment, the process is observed in state i, the time that it will sojourn in that state is independent of the time that has already been spent in it. Similarly, the next state to be entered depends only on the current state, and not on the time spent in it or on any previous states.

A CTMC also has an embedded DTMC (EMC), which describes the behaviour of the chain at state-transition instants, i.e. the probability that the next state is j, given that the current state is i [Dingle04a]. The EMC of a CTMC has a one-step transition matrix, $\boldsymbol{P}$, with entries

$$p_{ij} = \begin{cases} 0, & \text{if } i = j \\ \frac{q_{ij}}{-q_{ii}}, & \text{if } i \neq j. \end{cases} \tag{2.19}$$

Semi-Markov Processes

Definition A *semi-Markov process (SMP)* [Pyke61b, Howard71] is a generalisation of a Markov process, which allows generally distributed sojourn times. It changes states in the same way as a Markov process, but spends time in any state described by a random variable that depends on the state that the process currently occupies and on the state to which the next transition will be made. Hence, the memoryless property no longer applies to state sojourn times; however, at transition instants, SMPs behave like Markov processes, since the choice of the next state is only based on the current state.

Consider a Markov renewal process $\{(\chi_n, T_n) : n \geq 0\}$, where T_n is the time of the n^{th} transition and $\chi_n \in \mathcal{S}$ the state at the n^{th} transition. Let the kernel of this process be:

$$R(n,i,j,t) \;=\; \mathbb{P}(\chi_{n+1} = j, T_{n+1} - T_n \leq t \mid \chi_n = i) \tag{2.20}$$

where $i, j \in \mathcal{S}$. The continuous-time SMP $\{Z(t) : t \geq 0\}$, defined by the kernel R, is related to the Markov renewal process by:

$$Z(t) \;=\; \chi_{N(t)} \tag{2.21}$$

where $N(t) = \max\{n : T_n \leq t\}$ is the number of state transitions that have taken place by time t. Thus, $Z(t)$ represents the state of the system at time t. We consider only time-homogeneous SMPs in which $R(n,i,j,t)$ is independent of n:

$$\begin{aligned} R(i,j,t) &= \mathbb{P}(\chi_{n+1} = j, T_{n+1} - T_n \leq t \mid \chi_n = i) \quad \text{for any } n \geq 0 \\ &= p_{ij} H_{ij}(t) \end{aligned} \tag{2.22}$$

where $p_{ij} = \mathbb{P}(\chi_{n+1} = j \mid \chi_n = i)$ is the state transition probability between states i and j, and $H_{ij}(t) = \mathbb{P}(T_{n+1} - T_n \leq t \mid \chi_{n+1} = j, \chi_n = i)$ is the sojourn time distribution in state i when the next state is j. The state holding time is the amount of time that passes before making a transition from one state to another.

An SMP can thus be characterised by the matrices $\boldsymbol{P}$ and $\boldsymbol{H}$ with elements p_{ij} and H_{ij} respectively. In contrast, DTMCs have state holding times that are equal to a unit time (a step) and independent of the next state transition [Bradley06].

2.1.2 Stochastic Petri Nets

Petri Nets

Petri nets (PNs) [Petri62,Peterson77,Agerwala79,Peterson81,Reisig85,Murata89,Bause02] are a formalism for the description of concurrency and synchronisation in distributed systems. PNs have a convenient graphical representation (see Figure 2.4), which consists of the following components: *places*, drawn as circles, model conditions or resources. A place may represent a phase in the behaviour of a particular component, for example. Places may contain *tokens*, drawn as black dots, which are identity-less markers, and whose presence on a place indicates that the corresponding condition or local state holds. *Transitions*, drawn as rectangles, model activities within systems that effect a change in system state. Transitions are enabled and can fire (see Figure 2.5) when each of the places connected to it through unidirectional arcs contains at least one token. Tokens move between places according to the firing rules imposed by transitions. Upon firing, a transition removes a number of tokens from each of its predecessor places and deposits a number of tokens on each of its successor places. The number of tokens to remove and deposit by a transition are specified by annotations on its incoming and outgoing arcs. *Arcs* represent connections between places and transitions, and define the relationships between local states or conditions and events. An arc from a place to a transition indicates the local state in which the event can occur. An arc to a place from a transition indicates the local transformations that will be induced by the event [Hillston04].

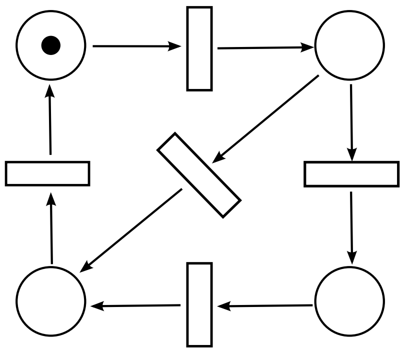


Figure 2.4: A Place-Transition net

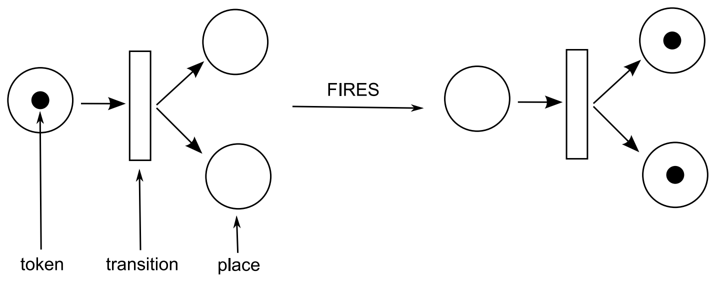


Figure 2.5: Firing of a Place-Transition net

The *state* (or *marking*) of a system modelled by a PN is identified by the number of tokens on each place in the net. A PN is defined by its structure and an initial distribution of tokens, the initial marking. The *reachability set* of a PN is the set of all possible markings that it may be in, having started from the initial marking and observing the firing rules. The simplest type of a PN is a Place-Transition net (P-T net):

Definition A *Place-Transition net* is a 5-tuple $PN = (P, T, I^+, I^-, M_0)$, where

- $P = p_1, \ldots, p_n$ is a finite and non-empty set of places,
- $T = t_1, \ldots, t_m$ is a finite and non-empty set of transitions,
- $P \cap T = \emptyset$,
- $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ are the backward and forward incidence functions,
- $M_0 : P \rightarrow \mathbb{N}_0$ is the initial marking.

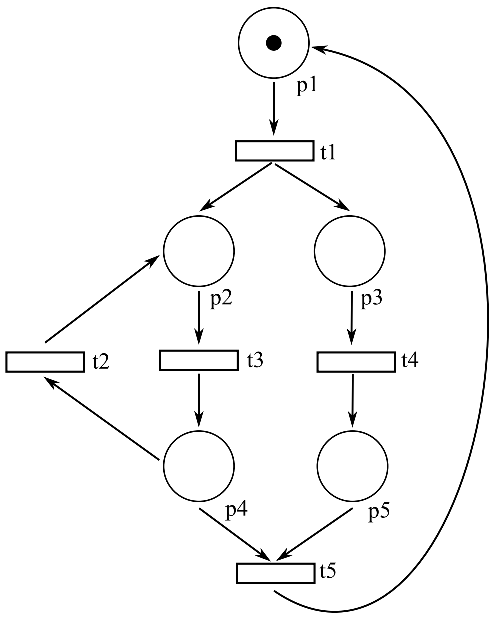
P-T nets are bipartite graphs, meaning that places can only be connected to transitions and vice versa. Backward and forward incidence functions specify the connection between places and transitions. If $I^-(p,t) > 0$, an arc leads from place p to transition t, hence, I^- is called the backward incidence function of transition t. It attaches a weight to the arc leading from p to t, which means that transition t is only enabled when place p contains at least as many tokens as specified by the weight. Firing destroys exactly this amount of tokens on p. Similarly, $I^+(p,t)$ specifies the number of tokens created on place p in case of firing t [Bause02].

Stochastic Petri Nets

PNs are useful in qualitative analysis, where functional behaviour is analysed, but since they do not incorporate any notion of time, PN-based performance analysis of systems is not possible. Quantitative performance analysis requires the temporal behaviour of systems to be represented by models. Therefore, traditional PNs have been extended to incorporate timing information. Among the most widespread of timed and stochastic extensions of Petri nets are *stochastic Petri nets (SPNs)* [Natkin81, Molloy81, Molloy82]. These are PN formalisms to which random variables have been added to represent the duration of activities or the delay until the occurrence of events.

Definition The continuous-time *stochastic Petri net* $SPN = (PN, \Lambda)$ is formed from the P-T net $PN = (P, T, I^-, I^+, M_0)$ by adding the set $\Lambda = (\lambda_1, \ldots, \lambda_m)$ to the definition. λ_i is the transition firing rate of transition t_i. The sojourn time in a state depends on which transitions are enabled, and is exponentially distributed with a parameter that is the sum of the individual firing rates of enabled transitions.

An example of a stochastic Petri net can be seen in Figure 2.6.



Transition	Rate
t_1	λ_1
t_2	λ_2
t_3	λ_3
t_4	λ_4
t_5	λ_5

Figure 2.6: A stochastic Petri net

Generating a Markov process from an SPN is simple, since the reachability graph of an SPN's underlying P-T net and the state-transition diagram of a Markov process are isomorphic, i.e. the number of states and the connection structure of both graphs are the same. However, many SPNs can result in Markov processes that have a very large number of states, which can make analysis infeasible [Bause02].

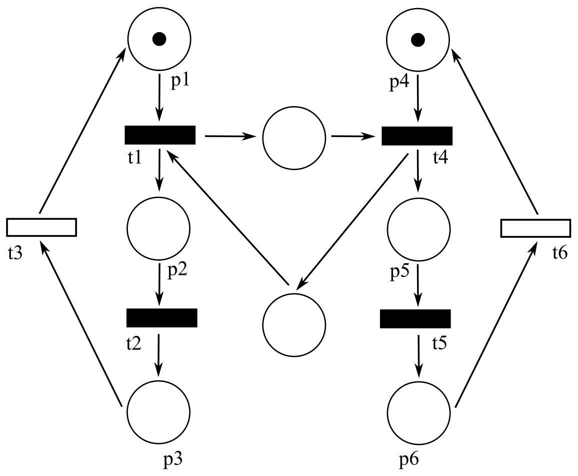
Generalised Stochastic Petri Nets

Generalised stochastic Petri nets (GSPNs) [Ajmone Marsan84,Ajmone Marsan95] extend SPNs by supporting immediate transitions, which fire in zero time upon being enabled, and timed transitions, which have an associated exponential delay. Enabled immediate transitions fire before timed transitions [Hillston04].

Definition A *generalised stochastic Petri net* is a 4-tuple $GSPN = (PN, T_1, T_2, W)$, where

- $PN = (P, T, I^-, I^+, M_0)$ is the underlying P-T net,
- $T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$,
- $T_2 \subset T$ denotes the set of immediate transitions, $T_1 \cap T_2 = \emptyset, T = T_1 \cup T_2$
- $W = (w_1, \ldots, w_{|T|})$ is an array where each $w_i \in \mathbb{R}^+$
  1. is a (possibly marking-dependent) rate of an exponential distribution specifying the firing delay when transition t_i is a timed transition, i.e. $t_i \in T_1$, or
  2. is a (possibly marking-dependent) weight, specifying the relative firing frequency when transition t_i is an immediate transition, i.e. $t \in T_2$

An example of a GSPN can be seen in Figure 2.7, which shows timed transitions as hollow rectangles and immediate transitions as filled rectangles.



Transition	Rate / Weight
t_1	w_1
t_2	w_2
t_3	λ_3
t_4	w_4
t_5	w_5
t_6	λ_6

Figure 2.7: A generalised stochastic Petri net

GSPNs do not directly describe a CTMC, since immediate transitions fire in zero time, and hence the sojourn time in markings that enable immediate transitions is not exponentially distributed. However, since the probability of changing from one marking to another is independent of the time spent in a marking, GSPNs describe semi-Markov processes. In models that contain immediate transitions, the reachability graph will contain instantaneous transitions that have no delay. Markings that enable immediate transitions are called *vanishing states*, because they are never observed, even though the stochastic process sometimes visits them. Markings that enable timed transitions are called *tangible states*, since the stochastic process sojourns in such markings for an exponentially distributed length of time [Bause02].

Semi-Markov Stochastic Petri Nets

Semi-Markov stochastic Petri nets (SM-SPNs) [Ciardo94, Bradley03b] are extensions of GSPNs that support arbitrary marking-dependent holding-time distributions, and that generate an underlying semi-Markov process rather than a Markov process.

Definition A *semi-Markov stochastic Petri net* is a 4-tuple $SMSPN = (PN, \mathcal{P}, \mathcal{W}, \mathcal{D})$, where

- $PN = (P, T, I^-, I^+, M_0)$ is the underlying P-T net,
- $\mathcal{P} : T \times \mathcal{M} \to \mathbb{Z}$, denoted $p_t(m)$, is a marking-dependent priority function for a transition,
- $\mathcal{W} : T \times \mathcal{M} \to \mathbb{R}^+$, denoted $w_t(m)$, is a marking-dependent weight function for a transition, to allow the implementation of probabilistic choice,
- $\mathcal{D} : T \times \mathcal{M} \to (\mathbb{R}^+ \to [0, 1])$, denoted $d_t(m, r)$, is a marking-dependent cdf for the firing time of a transition.

$\mathcal{M}$ is the set of all markings for a given net. In addition, SM-SPNs implement extended net-enabling functions:

- $\varepsilon_N : \mathcal{M} \to P(T)$ is a function that specifies net-enabled transitions of a given marking. A transition is net-enabled if all preceding places have occupying tokens.
- $\varepsilon_P : \mathcal{M} \to P(T)$ is a function that specifies priority-enabled transitions from a given marking. It selects only those net-enabled transitions that have the highest priority, i.e. $\varepsilon_P(m) = \{t \in \varepsilon_N(m) : p_t(m) = \max\{p_{t'}(m) : t' \in \varepsilon_N(m)\}\}$. For a

given priority-enabled transition, $t \in \varepsilon_P(m)$, the probability that it will be the one that actually fires after a delay sampled from its firing distribution $d_t(m, r)$, is

$$\mathbb{P}(t \in \varepsilon_P(m) \text{ fires}) = \frac{w_t(m)}{\sum_{t' \in \varepsilon_P(m)} w_{t'}(m)} \tag{2.23}$$

The choice of which priority-enabled transition is fired in any given marking is made by a probabilistic selection based on transition weights. This mechanism enables the underlying reachability graph of an SM-SPN to be mapped directly onto a semi-Markov chain.

SPNs and GSPNs can be easily expressed as SM-SPNs. To specify an SPN in the SM-SPN formalism, we let μ_t represent the exponential firing rate of a transition, t, in the SPN. Then we have:

- $p_t(m) = 0$ for all t, m,
- $w_t(m) = \mu_t$ for all m,
- $d_t(m, r) = 1 - e^{-\mu^{\Sigma} r}$, where $\mu^{\Sigma} = \sum_{t' \in \varepsilon_N(m)} \mu_{t'}$, i.e. the sum of the firing rates of the enabled transitions.

For GSPNs, the situation is very similar, except that the immediate transitions have priority over the timed transitions. The translation distinguishes between timed transitions ($t \in T_1$, having rate μ_t) and immediate transitions ($t \in T_2$, having a probabilistic weight c_t). Then we have

$$p_t(m) = \begin{cases} 0 & \textit{if}\ t \in T_1 \\ 1 & \textit{if}\ t \in T_2 \end{cases} \qquad w_t(m) = \begin{cases} \mu_t & \textit{if}\ t \in T_1 \\ c_t & \textit{if}\ t \in T_2 \end{cases} \tag{2.24}$$

$$d_t(m, r) = \begin{cases} 1 - e^{-\mu^{\Sigma} r} & \textit{if}\ t \in T_1 \\ H(r, 0) & \textit{if}\ t \in T_2 \end{cases} \tag{2.25}$$

where $\mu^{\Sigma} = \delta_{t \in \varepsilon_P(m)} \sum_{t' \in \varepsilon_N(m)} \mu_{t'}$ with $\delta_B = 1$ if condition B is true and 0 otherwise, and $H(r, a)$ is the Heaviside function with step time a.

2.1.3 Stochastic Process Algebras

Process algebras are abstract languages used for the specification and design of concurrent systems. Systems in process algebras are modelled as collections of entities, called agents, which execute actions. Actions are the building blocks of process algebras, and are used to describe sequential behaviours that may occur concurrently.

The development of *stochastic process algebras (SPAs)* was originally motivated by problems posed by the performance analysis of large computer and communication systems. The complexity of many modern systems results in very large and complex models. This is problematic both in terms of model construction and solution, and has led to an interest in alternative approaches to modelling that enable the creation of smaller models. Finding techniques for the solution of large Markov chains whose state spaces are finite but exceedingly large has been a major focus of performance analysis research for many years. Standard numerical techniques are not able to cope with very large models, which is why compositional approaches to model construction and solution that allow the separate solution of submodels have become popular. SPAs allow subsystems to be modelled separately, although models must be considered as single entities for the purposes of analysis. Subsystem models can be simplified in a way that the overall integrity of the model is not affected in order to make analysis feasible.

PEPA

The *Performance Evaluation Process Algebra (PEPA)* [Hillston94] was originally developed as a high-level description language for Markov processes. It extends classical process algebras by associating exponentially distributed delays with actions. An implicit choice is associated with each set of actions by the assumption of the race condition, which leads to a clear relationship between a process algebra model and a Markov process. In this manner, it is possible to extract performance measures from underlying Markov processes.

In PEPA, a system is described as an interaction of components that engage in activities. *Components* correspond to identifiable parts of the system, or roles in its behaviour, and represent the active units within a system. A component's behaviour is defined by the activities in which it can engage. PEPA uses the term *activity* to make a distinction between the usual process algebra notion of "instantaneous action" and the general behaviour of the system. Every activity in PEPA has an associated duration that is governed by an exponentially distributed random variable and an *action type*. Since an exponential distribution is uniquely determined by its parameter, the duration of an activity may be represented by a single real number parameter, called the *activity rate*. It may be any positive real number, or the distinguished symbol $\top$, which is treated as the unspecified rate. It is assumed that each discrete action within a system is uniquely typed, and that there is a countable set, $\mathcal{A}$, of all possible action types. Thus, the action types of a PEPA term correspond to the actions of the system being modelled. If there are several activities within a PEPA model that have the same action type, then they represent different instances of the same action by the system. There are situations when a system is carrying out some action (or sequence of actions), the identity of which is unknown or unimportant. To capture such situations, the distinguished action type τ is used. Activities of this type are private to the component in which they occur and are also not instantaneous. Each instance of an activity with a τ action type has an associated duration, just like any other type. However, unlike other types, multiple instances of τ-type activities within a PEPA model do not necessarily represent the same action by the system [Clark07].

The syntax of PEPA is defined by means of the following grammar:

$$P ::= (\alpha, r).P \mid P + Q \mid P \underset{L}{\bowtie} Q \mid P/L \mid A$$

An activity, $a \in Act$, is described by a pair (α, r), where $\alpha \in \mathcal{A}$ is the type of the action and $r \in \mathbb{R}^+$ is the parameter of the exponential distribution describing its duration. A small but powerful set of combinators is used to model complex behaviours [Hillston04]:

- *Prefix* (.): models the sequential behaviour of a component, which repeatedly undertakes one activity after another, to eventually return to the beginning of its behaviour.

- *Choice* (+): models a choice between two possible behaviours. This choice is represented as the sum of the possibilities. A race condition is assumed to govern the behaviour of simultaneously enabled actions. The continuous nature of the probability distributions ensures that the actions can not occur simultaneously. The rates of actions are chosen to reflect their relative probabilities.

- *Cooperation* ($\underset{L}{\bowtie}$): models scenarios where components need to synchronise over a set of actions. Actions in the cooperation set require the involvement of all cooperating components. The parallel combinator $||$ is used when multiple components behave completely independently and there is no cooperation between them. $||$ is equivalent to $\underset{\emptyset}{\bowtie}$.

- *Abstraction* (/): enables the hiding of actions, in order to make them private to the component(s) involved. The duration of hidden actions is not affected, but their type becomes hidden, represented as τ. Components cannot synchronise on τ.
- *Constant* (A): is a component whose meaning is given by a defining equation $A \stackrel{\text{def}}{=} P$, which gives the constant the behaviour of the component P. This is the mechanism of assigning names to components or behaviours.

One of the major advantages of PEPA over standard paradigms for specifying stochastic performance models is the inherent apparatus for reasoning about the structure and behaviour of models. The formality of the process algebra approach allows us to assign a precise meaning to every language expression, which implies that once we have a description of a given system, its behaviour can be deduced automatically. The semantics of PEPA associate a *derivation graph* with models, which describes all possible evolutions of every component. Derivation graphs are analogous to reachability graphs of GSPNs. Hence, the Markov process underlying a PEPA model can be obtained directly from the derivation graph. Components in PEPA correspond to states and activities correspond to transitions in the underlying CTMC. A state of the CTMC is associated with each node of the graph, and transitions between states are defined by considering the rates that are labelling arcs [Hillston04].

Solution techniques used to compute quantitative results for PEPA and GSPN models are identical – based on the numerical solution of the underlying CTMC. Starting from a PEPA (GSPN) model, the associated derivation (reachability) graph is obtained and reduced to the corresponding CTMC. This CTMC is then solved numerically to compute steady-state probabilities. State space explosion, however, is a major problem for models in both paradigms [Donatelli95].

2.1.4 Queueing Networks

A frequent application area for probability and stochastic processes is queueing theory. A *queue* consists of an arrival process, a buffer where customers await service, and one or more servers, representing a resource that is used by each customer for some period of time. Queues can be characterised by six attributes: the *arrival rate*, the *service rate*, the *number of servers*, the *capacity of the buffer*, the *customer population* and the *queueing discipline*. The first five of these characteristics may be represented concisely using Kendall's notation [Kendall53] for classifying queues. In this notation, a queue is represented as $A/S/c/m/N$:

- A stands for the customer arrival distribution. M denotes memoryless (exponential), G general and D deterministic distributions. Identifiers for other distributions may also be used.

- S represents the service time distribution. Service time is the time that a server spends serving a customer.

- c denotes the number of servers available to provide service to the queue.

- m specifies the capacity of the buffer, which customers need to join if a server is not available. The buffer capacity is assumed to be infinite by default. Customers who arrive when the buffer is full may be lost or blocked.

- N indicates the customer population, which is also infinite by default.

The last two classifiers may be omitted in the default case. The *queueing discipline* determines how a server selects a customer from the queue for next service. For example, the discipline might be *first-come-first-served (FCFS)*, which serves the customer who has been waiting for the longest time next, *last-come-first-served (LCFS)*, which ensures that the customer who has just joined the queue is served first, *random-selection-for-service (RSS)*, which selects customers for service from the queue at random, *priority (PRI)*, which assigns customers priorities that determine the order in which they will be served, or *processor sharing (PS)* which shares the service capacity by all customers in the queue [Cooper81, Hillston04].

If we are modelling interactions of devices that jobs visit sequentially, it is natural to model the system as a *queueing network (QN)*. A QN is a directed graph, in which devices or resources of the system are represented by nodes, called *service centres*, which themselves are queues. *Customers*, representing the jobs in the system, flow through the system and compete for its resources. Depending on the demand for resources and the service rate that customers experience, contention over a resource may arise, leading to the formation of a queue of waiting customers. It is assumed that after service at one service centre, customers progress to other service centres, following a pattern of behaviour that corresponds to tasks that they aim to achieve. The state of the system is typically represented by the number of customers occupying each of the service centres at a given point in time. *Arcs* in a QN represent the topology of the system, and together with *routing probabilities*, determine the paths that customers can take through the network. A network may be *open*, *closed* or *mixed*, depending on whether a fixed population of customers remain within the system or not. Customers may arrive from or depart to some external environment, or there may be classes of customers within the system that exhibit open and closed patterns of behaviour, respectively [Nelson95].

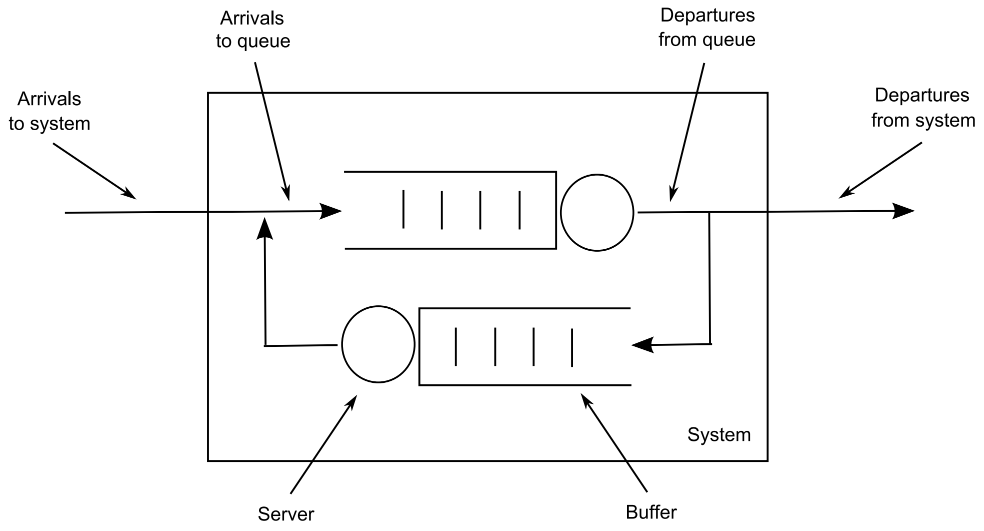


Figure 2.8: A simple open queueing network

A large class of queueing networks has been shown to have a straightforward and computationally efficient solution. Although this class excludes some interesting and important system features, when applicable, it allows performance measures to be derived without resorting to the underlying Markov process. The solution of such models, often termed *product form solutions*, allows individual queues within a network to be considered separately. Relatively simple algorithms exist for computing most performance measures based directly on the parameters of the queueing network. Performance questions of interest that may arise in such networks include the expected customer response time, the expected amount of time waiting for service or the maximum throughput of the system, for example.

2.2 Performance Query Specification

In the previous section, we have presented an overview of different formalisms that can be used for the modelling of real-life systems. This section discusses the next step in the analysis process: performance query specification. After the creation of a stochastic model, aspects of performance need to be specified with regards to which the modelled system is to be analysed. This is achieved by constructing a formal specification that defines performance properties and measures of interest that are to be extracted from the model. Such specifications are commonly referred to as *performance queries*.

A performance query is constructed in two stages. Initially, system designers or engineers who wish to analyse some system create performance queries intuitively in natural language. Once a query has been established in this manner, it needs to be translated into a formal representation. Such a formal representation needs to be able to express the concepts that appear in the query, and has to be understood by relevant analysis tools.

2.2.1 Performance Query Classification

Performance queries can be categorised in two ways. Firstly, a distinction can be made regarding the type of answer that a query is aiming to obtain. In this sense, queries can either be performance requirement or performance measure queries [Suto06b].

A *performance requirement* describes a property related to stochastic behaviour that must be satisfied by a system model. Such requirements have traditionally been phrased in terms of stochastic logic formulae, which can be verified by stochastic model checking tools. These tools establish whether or not requirements are satisfied by a model, and deliver appropriate *yes* / *no* answers.

A *performance measure* represents a quantitative measure that can be extracted from a model. Such measures can include for example response time distributions and densities, their convolutions, raw moments and percentiles, mean time to failure, system throughput, etc. Their evaluation is enabled by specialised quantitative analysis tools.

In addition, performance queries can be classified according to the concepts that they express. The three main categories that are relevant for the kinds of systems that we are considering are steady-state, transient state and passage time queries. Queries that do not fall into any of the above categories are classified as miscellaneous queries.

System states can be uniquely identified by state labels, which are atomic propositions that are associated with certain constraints on the state vector of the underlying model.

Steady-State Queries

Steady-state queries target the relative frequency of state occupancy for a set of states within a model. Long-run averages of resource-based metrics, such as availability or utilisation, can be expressed using steady-state measures. The idea of the long-run is based on the assumption that the system eventually reaches equilibrium. Examples of different types of steady-state query are as follows:

"What is the steady-state probability of the system being in any one of the states identified by labels $\{$'processing', 'processed', 'waiting'$\}$?"

This query aims to obtain a distribution that represents the long-term probability of a system being in a set of states, while the next query seeks a set of states that satisfy a specified constraint.

"Out of the set of states identified by labels $\{$'start', 'stop', 'error'$\}$, which have a steady-state probability that is greater than 0.12?"

A measure that can be indirectly derived from the steady-state probability distribution is the average rate of occurrence of an action. This is expressed in the following query:

"What is the productivity of the system, defined as the sum of the mean firing rate of action 'processed at A' multiplied by 100, and the average rate of occurrence of action 'processed at B' multiplied by 200?"

Transient State Queries

Transient state queries address the probability of a system being in a particular set of states at time t. They can be used to assess system reliability, since they are able to reason about the likelihood of systems entering a failure mode at a particular time, as shown in the following query:

"Is the probability that the system is in one of the error states identified by labels $\{$'aborted', 'failed'$\}$ at time instant 40 greater than 0.87?"

This is an example of a transient state requirement query, while the following is a transient state measure query that is interested in the states that satisfy certain transient state conditions, rather than in the probability with which the system is in a given set of states.

"What possible states can the system occupy at time instant 23 with probability exceeding 0.2?"

Passage Time Queries

The power to reason about response times is an essential ingredient in providing QoS guarantees in almost all concurrent and distributed systems, including mobile phone networks, Web and database servers, embedded systems, stock market trading platforms and health care systems. Passage time queries are typically useful for analysing system responsiveness, since they address the time that a system takes to move from one state to another, and reliability, since they are able to reason about time to failure scenarios.

For passage time queries, we refer to states that a passage can begin in as *start states*, while states that it terminates in are called *target states*. Certain specification formalisms used for passage time queries support multiple start and target states, in which case it is of interest to find the shortest time that a system requires to complete the passage between the two sets of states. In case of multiple start states, the passage from each possible start state is weighted by the relative probability of the passage beginning in that state. The following are examples of passage time queries:

"What is the distribution of time for the system to reach any one of the states identified by labels $\{$*'completed', 'aborted'*$\}$*, given that it has started in one of the states identified by labels* $\{$*'start', 'restart'*$\}$*?"*

This passage time measure query specifies a cdf as the performance measure of interest. The next query addresses the expected time that the system needs to complete the passage:

"What is the expected time of the system entering one of the error states identified by labels $\{$*'crashed', 'aborted'*$\}$*, given that it has started in one of the states identified by labels* $\{$*'ready', 'paused'*$\}$*?"*

The evaluation of this passage time measure query requires the calculation of the first moment of the passage time density. The following passage time requirement query attempts to verify whether a passage occurs between two sets of states within a given time with certain probability:

"Having started in one of the states identified by labels $\{$*'idle', 'initialised'*$\}$*, does the system enter any of the states identified by labels* $\{$*'processed', 'cancelled'*$\}$ *within 10 time units with probability between* 0.9 *and* 0.98*?"*

Miscellaneous Queries

It is possible to convolve multiple passages to obtain a single passage. Scenarios in which this might be applicable in practice are of the type when a system evolves from a state a to a state b, which is defined by the passage from a to b, and then evolves further from state b to state c, defined by the passage from b to c. The convolution of the two passages results in a single passage from a to c. This concept is demonstrated by the following query:

"What is the average time required to complete the passage defined by the convolution of the passage from the start state identified by label 'system ready' to the target state identified by label 'waiting' with the passage defined by the start state that is identified by label 'processing' and the target state that is identified by label 'idle'?"

Of additional interest to performance engineers is the ability to reason about moments of passage time densities and distributions in order to obtain expected values and variances for example. The following query illustrates these concepts:

"What is the variance of the passage time defined over the start state that is identified by label 'new customer' and the target states identified by labels $\{$*'customer processed'*, *'customer left'*$\}$*?"*

Queries relating to response time quantiles are particularly important, since they are increasingly used as key QoS metrics in Service Level Agreements. The following query, which corresponds to a UK Government target, involves a response time quantile:

"Is it true that patients of an Accident & Emergency unit are seen, treated and discharged in under 4 hours 98% *of the time?"*

Queries can be extended to include further restrictions. Since a change of state in a system is effected by an action, it may be of interest to specify a set of *included* or *excluded actions*, in order to observe the dynamic behaviour of the system, provided that certain actions do or do not occur. For example:

"What is the probability of the passage from the start state identified by label 'customer arrived' to the target state identified by label 'customer left' completing in 71 *time units, provided that action 'customer pays' occurs and that action 'customer does not pay' does not occur along the passage?"*

Passage time queries can be constrained even further by the requirement that a given set of states should be avoided during the passage. Such states are termed *excluded states*. The following query incorporates such a constraint:

"What is the probability of the passage from the start state identified by label 'patient admitted' to the target state identified by the label 'patient recovered' completing in 71 *time units, provided that the states identified by labels* $\{$*'patient comatose', 'patient's heart stopped'*$\}$ *are avoided along the passage?"*

Performance queries can consist of multiple performance requirements and measures, and independent performance queries can be composed together into a single query. Such compound queries could take the following form:

"Is the probability of a passage from the start state identified by label 'item ordered' to the target state identified by label 'item delivered' completing in 50 *time units less than* 0.88*, and what is the density of time that it takes to complete this passage?"*

2.2.2 Logical Specification Formalisms

Probabilistic techniques, and probabilistic logics in particular, have been popular in the past for the specification and verification of properties of systems that exhibit uncertainty. Probabilistic logics are widely used in model checking (see Section 2.3.1), due to their ability to express relevant performance properties concisely, rigorously and in a verifiable manner, while also supporting the composition of simple queries into more complex ones.

Continuous-time probabilistic logics come in many different flavours. Among them are *CSL* [Aziz96, Aziz00, Baier00, Katoen01, Baier03], *aCSL* [Hermanns00] and *asCSL* [Baier04] for Markovian models, *CSRL* [Baier00] for Markov reward models, *CSL* and *eCSL* [Bradley03c] for semi-Markov models, and *aCSL* for process algebras.

Many of these logics are popular in academic circles; however, their use in industry is limited – mostly because of their obfuscating nature, steep learning curve and limitations in terms of expressiveness with regards to quantitative performance properties. These limitations are a result of stochastic logics only being able to express performance requirement queries, but not performance measure queries. Depending on the variant of stochastic logic, they are also constrained to reasoning about state-, path-, action- and reward-based concepts only. Below, we present an overview of two major stochastic logic variants.

CSL

CSL, the Continuous Stochastic Logic, is able to express performance measures by selecting states and paths from Markovian systems that meet steady-state and passage time criteria. CSL operates on CTMCs on the state level, and expresses performance requirements as formulae. These can be of two types: state formulae are *true* or *false* in a specific state, while path formulae are *true* or *false* along a specific path of the underlying model. The logic has the ability to express steady-state, path-based and nested constraints. The syntax for these constructs is as follows:

$$\begin{aligned}
\sigma &\overset{\text{def}}{=} tt \mid a \mid \neg\sigma \mid \sigma \wedge \sigma \mid \mathcal{S}_{\boldsymbol{\rho}}(\sigma) \mid \mathcal{P}_{\boldsymbol{\rho}}(\varphi) \\
\varphi &\overset{\text{def}}{=} \mathcal{X}^{\boldsymbol{\tau}}\sigma \mid \sigma\,\mathcal{U}^{\boldsymbol{\tau}}\sigma
\end{aligned}$$

tt represents a truth value, while atomic proposition $a \in AP$ (AP being the set of atomic propositions) holds in state σ if σ is labelled with a. $\mathcal{S}_{\boldsymbol{\rho}}(\sigma)$ asserts that the aggregate steady-state probability for the states satisfying σ lies in $\boldsymbol{\rho}$, whereas $\mathcal{P}_{\boldsymbol{\rho}}(\varphi)$ expresses a constraint on the probability to lie in the range $\boldsymbol{\rho}$ with which paths satisfy φ. Paths are defined by φ and can take the form $\mathcal{X}^{\boldsymbol{\tau}}\sigma$ or $\sigma_1\mathcal{U}^{\boldsymbol{\tau}}\sigma_2$. The $\mathcal{X}^{\boldsymbol{\tau}}\sigma$ path formula asserts that a transition is made to a σ state at some time, $t \in \boldsymbol{\tau}$, while $\sigma_1\,\mathcal{U}^{\boldsymbol{\tau}}\sigma_2$ asserts that σ_2 is satisfied at some time instant within the interval $\boldsymbol{\tau}$, while σ_1 holds at all preceding time instants.

The semantics of the logic are expressed by stating the conditions under which a single state s satisfies each clause of a σ-formula. This is expressed by the satisfiability relation $s \models \sigma$. The clause a is a label, and a state s satisfies that label if $a \in \mathcal{L}(s)$. Thus, using the negation and conjunction clauses in combination with labelling allows whole sets of states to be defined with a σ-formula. The set of states specified in this manner is written $\mathrm{Sat}(\sigma) = \{s \in S \mid s \models \sigma\}$. The formal semantics of CSL are defined as [Baier03]:

$$\begin{array}{lcll}
s & \models & tt & \text{for all } s \\
s & \models & a & \text{iff } a \in \mathcal{L}(s) \\
s & \models & \neg\sigma & \text{iff } s \not\models \sigma \\
s & \models & \sigma_1 \wedge \sigma_2 & \text{iff } s \models \sigma_1 \wedge s \models \sigma_2 \\
s & \models & \mathcal{S}_{\boldsymbol{\rho}}(\sigma) & \text{iff } \Pi_J \in \boldsymbol{\rho} \quad \text{where } J = \mathrm{Sat}(\sigma) \\
s & \models & \mathcal{P}_{\boldsymbol{\rho}}(\varphi) & \text{iff } \mathbb{P}(\sigma \in \mathrm{Path}(s) \mid \sigma \models \varphi) \in \boldsymbol{\rho}
\end{array}$$

where Π_J is the steady-state probability of being in any of the states in J, and $\mathrm{Path}(s)$ is the set of all paths starting from s. Further, a path ψ satisfies a path formula, φ, as follows:

$$\begin{array}{lcll}
\psi & \models & \mathcal{X}\sigma & \text{iff } \exists\psi[1] \models \sigma \\
\psi & \models & \sigma_1\,\mathcal{U}^{\boldsymbol{\tau}}\sigma_2 & \text{iff } \exists t \in \boldsymbol{\tau}\ .(\psi@t \models \sigma_2\ \wedge \forall t' < t, \psi@t' \models \sigma_1)
\end{array}$$

where $\psi[1]$ is a state immediately succeeding the start state of ψ; $\psi@t$ is the state that the system is in at time t on the path ψ. The $\mathcal{X}$ path operator is often referred to as the 'Next State' operator and asserts that the next transition will be made to a σ state. The time-bounded Until formula $\sigma_1\,\mathcal{U}^{\boldsymbol{\tau}}\sigma_2$ asserts that σ_2 is satisfied at some time instant within the interval $\boldsymbol{\tau}$ and that σ_1 holds at all preceding time instants.

To illustrate how a performance query is represented in CSL, consider the following:

"Starting from state s_3, is the probability of a 3-processor system being down within 10 time units after having continuously operated with at least two processors at most 1%?"

In CSL, this query is expressed by the following formula:

$$s_3 \models \mathcal{P}_{\leq 0.01}((up_3 \vee up_2)\,\mathcal{U}^{[0,10]}\,down)$$

An example of nested constraints is demonstrated in the following scenario. A robot is moving in an $n \times n$ grid from the bottom left corner to the top right corner. Let c be a Boolean state variable that is *true* when the robot is communicating, and let x and y be two integer-valued state variables recording the current location of the robot [Younes05]. We want to express and subsequently verify the following property:

"Does the robot reach the top right corner of the grid within 100 time units, with probability at least 0.9, while maintaining at least a 0.5 probability of periodically (every 9 time units) communicating with a base station?".

This is expressed in CSL with the following formula:

$$\mathcal{P}_{\geq 0.9}(\mathcal{P}_{\geq 0.5}(tt\,\mathcal{U}^{[0,9]}\,communicate)\,\mathcal{U}^{[0,100]}\,corner)$$

eCSL

eCSL, the extended Continuous Stochastic Logic, is an extension of CSL that operates on SM-SPNs. In contrast to other logical formalisms, eCSL operates at the model level, rather than at the state-transition level. It was designed to express a broader spectrum of performance requirements than CSL, including a richer class of passage time quantities and constraints on transient state distributions. eCSL does not support compound formulae in order to simplify the representation mechanism, and introduces separate layers for the specification of sets of states and of performance criteria. The power of eCSL lies in its ability to express, in a single compound logical formula, the reliability, availability and response time requirements of semi-Markovian systems. The syntax of eCSL is as follows:

$$\begin{array}{lcl} \sigma & \stackrel{\text{def}}{=} & tt \;|\; \neg\sigma \;|\; \sigma \wedge \sigma \;|\; p[N] \\ \varphi & \stackrel{\text{def}}{=} & tt \;|\; \neg\varphi \;|\; \varphi \wedge \varphi \;|\; \mathcal{S}_\rho(\sigma) \;|\; \mathcal{T}_\rho^\tau(\sigma,\,\sigma) \;|\; \mathcal{P}_\rho^\tau(\sigma,\,\sigma) \end{array}$$

$p[N]$ identifies a marking of the SM-SPN by specifying the constraint that place p contains N tokens. $\mathcal{S}_\rho(\sigma)$ holds if the steady-state probability of occupying the set of states identified by σ lies in the range ρ. The formula $\mathcal{T}_\rho^\tau(\sigma_1,\ \sigma_2)$ is satisfied by a set of start states if the probability of occupying states σ_1 at time t, while not having visited states in the set denoted by σ_2, lies in ρ for all times $t \in \tau$. The expression $\mathcal{P}_{\rho}^{\tau}(\sigma_1,\ \sigma_2)$ holds for a set of start states if the time taken to complete the passage to the set of target states identified by σ_1, while not having passed through the set of states marked by σ_2, lies in the range τ with probability $p \in \rho$.

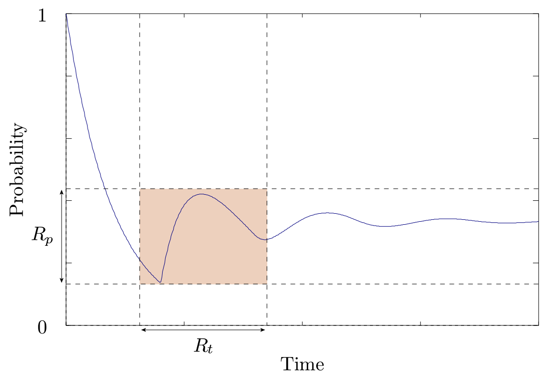


Figure 2.9: A graphical representation of a logical constraint on system reliability, using a $\mathcal{T}$-formula [Bradley03c]

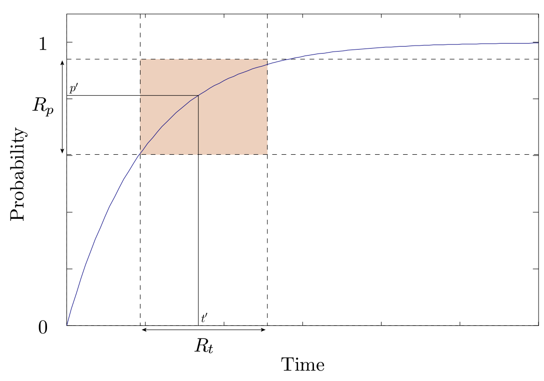


Figure 2.10: A graphical representation of a logical constraint on response-time, using a $\mathcal{P}$-formula [Bradley03c]

Figure 2.9 shows an example of a reliability constraint $\mathcal{T}_{R_p}^{R_t}(\sigma_1, \sigma_2)$ that reasons about the transient distribution in the shaded area. It expresses the requirement that the probability of system failure should lie within the region R_p over the time region R_t. If any part of the transient function over R_t lies outside the region R_p, the property fails. Figure 2.10 shows an example response time constraint $\mathcal{P}_{R_p}^{R_t}(\sigma_1, \sigma_2)$ which reasons about a passage time distribution. It expresses the requirement that there should exist a response time range $t' \in R_t$ with probability of occurrence $p' \in R_p$. This is applied to the cdf of a passage time; thus if the shaded region does not intersect the cdf, the property is not satisfied.

As an example of how eCSL is applied, consider the following query:

"Does the system reach the set of states that is identified by 175 tokens on place p_2 within 10 seconds with at least 90% probability, given that it has started from one of the states that is identified by 35 tokens on place p_1 and 10 tokens on place p_5, and has not reached any state in the set of states that are identified by having 1 token on place p_6?"

eCSL expresses this query with the following formula:

$$\text{Sat}(p_1[35] \wedge p_5[10]) \models \mathcal{P}_{(0.9,1]}^{[0,10)}(p_2[175], p_6[1])$$

The $p_n[m]$ expressions define sets of states on the SM-SPN model. For instance, the constraint $p_1[35] \wedge p_5[10]$ selects all markings of the model that have 35 tokens on place p_1 and 10 tokens on place p_5. Simple formulae can be composed to form compound queries, which can be verified on a model. The generic formula below shows how formulae can be defined, in which availability, reliability and response time properties must hold simultaneously:

$$\vec{m} \models \underbrace{\mathcal{S}_{P_1}(\sigma_1)}_{\text{availability}} \wedge \underbrace{\mathcal{T}_{P_2}^{T_2}(\sigma_2, \sigma_3)}_{\text{reliability}} \wedge \underbrace{\mathcal{P}_{P_3}^{T_3}(\sigma_4, \sigma_5)}_{\text{response time}}$$

2.2.3 Graphical Approaches

To our knowledge, no graphical formalism has been developed so far for performance query specification on stochastic system models.

The closest we can get to such a formalism, purely in terms of graphical specification ability, is L_R [Lee97], a specification language for high-level *behavioural properties* of real-time systems. Note that L_R is only able to address behavioural properties of systems, and not performance properties; however, its underlying concept of representing system properties graphically is similar to what one might expect to see in a graphical query specification language for performance properties. [Lee97] identifies a common drawback of property specification with stochastic logics, namely that formulae representing properties of even moderate complexity are generally hard to understand, and hence, usually only experts in formal methods are able to apply them properly. L_R has been developed as a potential solution to this drawback. The language is able to express queries on system models defined in the real-time process algebra ACSR [Lee94] by using a two-level query construction mechanism. Experts create patterns of partial stochastic logic formulae, which are then mapped onto graphical templates that hide certain details from users. Users specify properties of their systems in graphical queries through the use of these templates, and subsequently, templates are translated into their corresponding stochastic logic equivalents during query evaluation.

To illustrate the application of L_R, consider the following behavioural property: *After event a, we observe event b, and then event c. We require that b happens between 2 and 5 time units after a, and that c happens between 4 and 10 time units after b, and between 3 and 10 time units after a.* In the stochastic logic TCTL [Alur91], this property would be expressed by the following formula:

$$\Box a \Rightarrow x.\Diamond(b \Rightarrow y.(x \in [2,5] \wedge \Diamond(c \wedge (x \in [3,10] \wedge y \in [4,10]))))$$

This is clearly a rather complex expression, which is only meaningful to experts. L_R facilitates the expression of the same property with an equivalent, yet simpler representation, as shown in Figure 2.11.

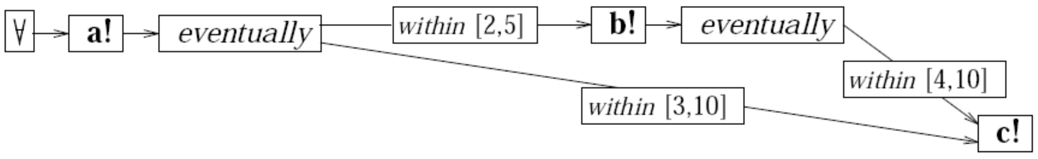


Figure 2.11: An L_R representation of the example TCTL query

There exist various approaches to graphical *performance property* specification; however, these are mostly semi-graphical in nature and formalisms that only support performance property annotations on system specifications, rather than the ability to specify performance queries directly. These approaches are based on UML, the Unified Modelling Language [OMG07].

[López-Grao04] uses *UML activity diagrams for system representation and the SPT profile for annotating performance requirements*. Two kinds of UML behavioural diagrams are particularly applicable to performance modelling: statechart diagrams, which model the life cycle of objects in the system, and activity diagrams, which characterise system behaviour by describing activities. Activity diagrams are specialisations of UML state machines, whose main purpose is the expression of the internal control flow of a process, as opposed to statechart diagrams which are often driven by external events. Combining the use of statecharts and activity diagrams, all paths of the potential system dynamics can be modelled. In this approach, the SPT `<<PAprob>>` and `<<PArespTime>>` tags are used, which allow the annotation of routing rates and action durations, respectively. Such annotations are attached to transitions in activity diagrams, in order to allow the assignment of different action durations that depend on a decision. Time annotations are added to the model wherever an action is executed, and probability annotations are supplied whenever a decision needs to be made, such as in the presence of guard conditions for example. Systems are modelled by means of the two kinds of diagrams, and performance requirements are specified according to the SPT profile. Each diagram is then translated automatically into a labelled GSPN (LGSPN), an extension of the GSPN formalism. LGSPN models are subsequently composed, in order to obtain a single analysable performance model of the system for a particular scenario, which is determined by the diagrams that have been modelled. These scenario description models can then be analysed or simulated using various well-established GSPN tools to obtain performance metrics of interest.

[Jansen05] uses *StoCharts, a QoS-oriented extension of UML statechart diagrams*, which enhances the basic statechart formalism with general time delays and probabilistic choice, to build stochastic automata that can be analysed by simulation. The two extensions have been added to provide software engineers with a simple way to describe probabilistic properties of stochastic systems. System models created with StoCharts can be reduced to generalised SMPs (GSMPs), which are amenable to discrete-event simulation-based evaluation. If only exponentially distributed delays are used, the models can be reduced to CTMCs, which can then be solved with respect to steady-state and transient measures, using efficient numerical solution techniques. Stochastic model checking is another method that can be applied to the resulting GSMP or CTMC models, in order to verify performance-related properties. An example StoCharts system specification is shown in Figure 2.12. The example shows the workflow of a car damage assessor, who assesses on behalf of an insurance company whether a damaged car should be repaired or not and whether a garage offers an acceptable price for the repairs. Timing delays in the model are indicated by $after$ annotations on transitions, and parallelism is demonstrated by the states $Repair$ and $Report$.

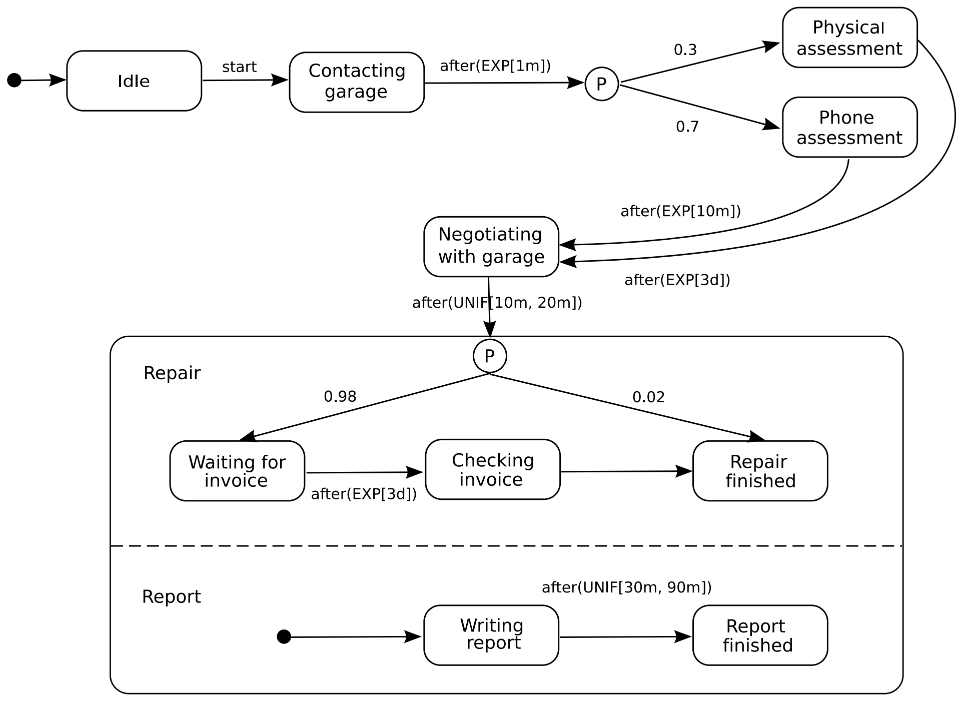


Figure 2.12: A StoChart example

[López-Grao04] and [Jansen05] represent the latest developments in graphical performance property specification, and clearly highlight the need for a new formalism that is not only able to reason about a wide range of performance concepts, but that is also completely independent of underlying system specifications.

2.2.4 Tool-specific Specification Languages

Most quantitative analysis tools combine the ability to model probabilistic systems and to analyse resulting models according to a set of supported performance measures and criteria. Many tools implement their own proprietary languages for performance query specification; therefore we provide below a brief overview of the tool languages of a representative range of analysis tools.

The *DNAmaca* Specification Language

The *DNAmaca* specification language [Knottenbelt96], used by the *DNAmaca*, *SMARTA* and *HYDRA* tools (see Section 2.4.1), provides a flexible high-level model description that is used by a state space generator as a basis for the construction of a CTMC of the model. Functional properties, such as invariants, that are to be checked during the state generation process can be specified with general C++ expressions, in addition to a variety of performance measures relating to model states and transitions.

Model Specification

A *model description* specifies the initial state of a system, the components of a general state, and the conditions on and effects of transitions between states.

```
model_description = \model {
  {state_vector | initial_state |  transition_declaration |
   constant | help_value | invariant | state_output_function |
   primary_hash_function | secondary_hash_function |
   additional_headers}*}
```

The *state description vector* consists of a set of components that together describe a state of the system. Each unique value of these components corresponds to a state. An *initial state* must be specified for reachability analysis purposes through assignments to the components of the state vector.

```
state_vector = \statevector{
  {<variable type> <identifier> {,<identifier>}*;}*
}

initial_state = \initialstate{{<assignment>}*}
```

Transitions describe how a system changes state. Possible transitions from a particular state are specified by describing enabling conditions, involving elements of the state vector, an action to be taken if the transition has executed, a rate for timed transitions or relative weight for instantaneous transitions, and an optional priority, which allows transitions of the same kind (immediate or timed) to take precedence over one another.

```
transition_declaration = \transition{<identifier>}{
  \condition{<boolean expession>}
  \action{{<assignment>}*}
  \rate{<real expression>} | \weight{<real expression>}
  \priority{<non-negative integer>}
}
```

Performance Measure Specification

Performance results provide a mapping from low-level results like state probabilities and transition rates to higher-level concepts, such as throughput or mean buffer occupancy. *Performance measures* can generally be classified as state or count measures.

```
performance_measures = \performance{
  {state_measure | count_measure}*
}

state_measure = \statemeasure{<identifier>}{
  \estimator{{mean | variance | stddev |  distribution}*}
  \expression{<real_expression>}
}

count_measure = \countmeasure{<identifier>}{
  \estimator{mean}
  \precondition{<boolean_expression>}
  \postcondition{<boolean_expression>}
  \transition{all | {<identifier>}*}
}
```

A *state measure* is used to determine the mean and variance of a real expression that is defined at every state in the system, e.g. the average number of tokens on a particular place of a GSPN or some transition's enabling probability. The mean, variance, standard deviation and distribution of state measures can be computed. A *count measure* is used to determine the mean rate at which a particular event occurs, e.g. the rate at which a transition fires. The occurrence of an event is specified by a precondition on the current state, a postcondition on the next state, and transitions that fire during the transition from the current to the next state.

The *HYDRA* Specification Language

The specification language of the *HYDRA* tool [Dingle04a] is an extended version of the *DNAmaca* specification language. Syntax has been added to allow the expression of first passage time and transient performance measures. For *passage time* queries, users need to specify conditions that identify the source and target states of the passage, as well as the time range to which the calculation should be restricted. The time range is specified by an initial value t, an incremental step and a maximum value. The source and target conditions are expressed as Boolean expressions in terms of the elements of the model's state vector. Conditions for *transient* measures are expressed in a similar fashion.

```
passage_time_measure = \passage{
  \sourcecondition{<Boolean expression>}
  \targetcondition{<Boolean expression>}
  \t_start{<real expression>}
  \t_stop{<real expression>}
  \t_step{<real expression>}
}

transient_state_measure = \transient{
  \sourcecondition{<Boolean expression>}
  \targetcondition{<Boolean expression>}
  \t_start{<real expression>}
  \t_stop{<real expression>}
  \t_step{<real expression>}
}
```

The *SMARTA* Specification Language

The *SMARTA* specification language [Dingle04a] is an extension of *DNAmaca*'s specification language that, in order to express semi-Markov chains, allows the specification of transitions whose state holding times are not constrained to an exponential distribution. The specification language was designed for the description of SM-SPNs, but is able to specify any semi-Markov chain.

Transitions are specified in the same way as weighted transitions in the *DNAmaca* specification language, but it is in addition also necessary to describe the firing time density function associated with each transition. These density functions are described in terms of their Laplace transforms. Users are also required to provide a C++ function that returns the value of the firing time density function's Laplace transform at a given s-value. Several macros are defined by default that encode the Laplace transforms of common firing time distributions. Steady-state specification is as for *DNAmaca*, and passage time specification is as for *HYDRA*.

```
transition_declaration = \transition{<identifier>}{
  \condition{<Boolean expression>}
  \action{{<assignment>}*}
  \weight{<real expression>}
  \priority{<non-negative integer>}
  \sojourntimeLT{<function>}
}
```

The PRISM Specification Language

The PRISM tool (see Section 2.4.1) uses its own proprietary language for the expression of properties on DTMCs, CTMCs and Markov decision processes. The language is very reminiscent of CSL, and its basic syntax is defined by the following grammar:

```
prop     ::= true | false | expr | !prop | prop & prop |
             prop | prop | prop => prop |
             P bound [pathprop] | S bound [prop]

bound    ::= <p | <=p | >=p | >p |

pathprop ::= X prop | prop U prop | prop U time prop |
             F prop | F time prop | G prop | G time prop

time     ::= >=t | <=t | [t,t]
```

where `expr` evaluates to a Boolean; `P bound [pathprop]` evaluates to *true* if the probability with which `pathprop` is satisfied lies within the bound represented by `bound`;

`S bound [pathprop]` evaluates to *true* if the steady-state probability of `prop` lies within the bound represented by `bound`; `p` evaluates to a double in the range $[0, 1]$; `X prop` evaluates to *true* if `prop` holds in the next state; `prop1 U prop2` evaluates to *true* if `prop1` holds throughout until `prop2` holds; `prop1 U prop2` evaluates to *true* if `prop1` holds throughout within the time constraint specified by `time`, after which `prop2` holds; `F prop` evaluates to *true* if `prop` eventually holds; `F time prop` evaluates to *true* if `prop` eventually holds within the time constraint specified by `time`; `G prop` evaluates to *true* if `prop` always holds; `G time prop` evaluates to *true* if `prop` always holds within the time constraint specified by `time`; and `t` evaluates to a non-negative double or integer.

PRISM identifies states of the model that correspond to certain situations by specifying an expression that evaluates to a Boolean value. Such an expression typically contains references to variables and constants from the model to which it relates. The states corresponding to this expression are those for which the expression evaluates to *true*. A property is evaluated with respect to a single state of a model. For the syntax given above, all properties evaluate to Boolean values, i.e. for any model state s, a property is either true and hence satisfied, or false and hence not satisfied. For the basic operators of the logic (i.e. `true, false, expr, !, &, |, =>`) the semantics are as for classical propositional logic:

- `true` is true in all states,
- `false` is not true in any state,
- `expr` is true if the PRISM expression `expr` evaluates to true,
- `!prop` is true if `prop` is not true,
- `prop1 & prop2` is true if both `prop1` and `prop2` are true,
- `prop1 | prop2` is true if either `prop1` or `prop2` is true,
- `prop1 => prop2` is true if `prop1` implies `prop2`.

PRISM is able to specify properties of the following types:

“From an initial state, is the probability that more than 5 errors occur within the first 100 time units less than 0.1?”

```
"init" => P<0.1 [ F<=100 num_errors > 5 ]
```

“When a shutdown occurs, is the probability of system recovery being completed in between 1 and 2 hours without further failures occurring greater than 0.75?”

```
"down" => P>0.75 [ !"fail" U[1,2] "up" ]
```

“In the long-run, is the probability that an inadequate number of sensors are operational less than 0.01?”

```
S<0.01 [ num_sensors < min_sensors ]
```

“What is the probability that process 1 terminates before process 2 does?”

```
P=? [ !proc2_terminate U proc1_terminate ]
```

“What is the long-run probability of the queue being more than 75% full?”

```
S=? [ queue_size / max_size > 0.75 ]
```

PRISM allows the computation of the values of such properties for a range of parameters and plot graphs of the results using experiments.

2.2.5 Comparison of Techniques

Logical Specification Formalisms

Logical property specification formalisms define performance properties over stochastic models. They do not characterise systems themselves, as they are not modelling formalisms. They are processed by model checking tools that carry out a verification of their performance specifications on models. These verifications can only either evaluate to *yes* or *no*.

In addition to these common characteristics, each stochastic logic variant has its own specialty. *CSL* expresses state- and path-based steady-state and passage time properties on the state level of CTMCs, while *eCSL* is able to express steady-state, transient state and passage time properties on the model level of SM-SPNs. It attempts to ease the property specification process by aligning it with a high-level modelling formalism. However, it is not able to reason about system actions.

Logical specification formalisms were designed for the very specific purpose of model checking. Hence, their expressiveness is constrained by their area of application. Unlike most graphical specification languages, they do not combine model specification and performance annotations, but rather only operate as performance property specification mechanisms. They allow sophisticated and complex performance requirement queries to be formulated, which is not possible with graphical formalisms. In addition, they enable query specification through concise formulae, whereas in contrast, graphical languages can be complex and confusing, and generally need to be reduced to some intermediary representation before they can be used for analysis.

Graphical Approaches

[Lee97] presents L_R, a graphical specification language for behavioural properties, which attempts to alleviate the challenges involved in specifying such properties in stochastic logics. It allows users to construct queries using high-level graphical templates. Queries are eventually translated into stochastic logic for evaluation purposes. The strength of this approach is its graphical specification mechanism and expressive power. It's weakness is that it is dependent on stochastic logics and that it is only able to reason about behavioural properties.

[López-Grao04] adapts the SPT profile to UML activity diagrams. Performance information is integrated into the UML model description, as in the previous case, and models are converted into LGSPNs, which can be solved for steady-state and transient measures in the traditional manner. The same advantage and disadvantage applies as before.

[Jansen05]'s specification of system models with *StoCharts* contains additional timing information and probabilistic choice. Similarly to the previous two approaches, performance information is incorporated into the model. Models are reduced to GSMPs that allow steady-state and transient state solutions of systems. Similar arguments with regards to expressiveness apply as before.

Tool-Specific Languages

The *DNAmaca* specification language and its extensions for *HYDRA* and *SMARTA* describe models and performance measures concisely in C++ syntax. The main purpose of *DNAmaca*'s specification language is the concise description of stochastic system models. In addition, it provides users with the ability to specify a small number of performance measures, which are to be obtained from the model. The language extensions for *HYDRA* and *SMARTA* provide support for the specification of additional performance measures. Since *DNAmaca* , *HYDRA* and *SMARTA* are command-line tools, their specification languages were primarily designed with functionality, rather than with user-friendliness in mind. However, they are certainly powerful with regards to their intended purpose.

In contrast, the simplicity and relative ease of use of PRISM's property specification language makes it appealing to a large audience. It has syntactic similarities to stochastic logics, but is clearer and more intuitive to understand and use. As a result, it has also been incorporated into other tools. A great advantage lies in its extensibility, since its syntax can be extended to cater for additional properties that are to be integrated into tools in the future. This is a very important characteristic, which for example stochastic logics do not possess due to their syntactic restriction. Another advantage is the language's concise and rigorous nature, which is derived from stochastic logics, its original inspiration. However, due to the similarity to stochastic logics, users have to become familiar with the language before being able to use it comfortably and effectively.

When contrasting tool-specific languages with stochastic logics, one usually finds that tool-specific languages either tend to be in some form based on stochastic logics, or that they share no commonalities with them. PRISM's property specification language is a good example of one that was inspired by stochastic logics, and *DNAmaca*'s specification language that of one that was not. Tool-specific languages often tend to be more intuitive than stochastic logics, since they have more freedom in the specification of properties, and as such can make the syntax more digestible to users. Model checkers that use stochastic logics for property specification purposes invariably have their own tool-specific languages for specifying stochastic logic formulae.

At present, tool-specific languages have an inherent advantage when compared to graphical approaches in that they allow for greater expressiveness, due to their ability to specify as much detail as needed, whereas graphical approaches are bound to performance annotations on system models that impose a serious restriction on their level of expressiveness.

2.3 Techniques of Performance Analysis

In the previous sections, we have seen popular methods used in performance modelling for the mathematical representation of real-life systems, and approaches to the specification of performance queries on stochastic system models. This section provides an overview of the most widely-used methods for the evaluation of performance queries on stochastic models.

2.3.1 Probabilistic Model Checking

Verification is the process of ensuring the correctness of systems. It is a major challenge in the process of system development, and hence also an important part of performance analysis. Simulation and testing are the two most widely-used methods for system verification, but in the case of complex asynchronous systems, these techniques are only able to analyse a limited number of possible behaviours. Simulations and test runs are often very time-consuming and may need to be carried out a number of times before informative conclusions can be reached [Clarke Jr.01].

An attractive alternative is *formal verification*, which carries out an exhaustive analysis of all possible behaviours of a system, and leaves no errors or design flaws undiscovered. There are numerous approaches to formal verification; however, model checking in particular has found widespread acclaim and adoption. *Model checking* is the process of verifying a behavioural property of a system over a given model through the exhaustive enumeration of all reachable system states and the behaviours that result in them. Compared to other approaches, model checking has a number of distinct advantages:

- The verification process is fully automated and requires no user intervention. Users are only required to provide a description of a system in the form of a stochastic model, together with a performance query that specifies properties to be checked. Equipped with these, a model checker is able to perform computations to obtain results indicating whether or not the properties are satisfied by the model.
- If properties are satisfied, the model checker terminates with the answer *true*. Otherwise, a counterexample is produced, which highlights a scenario where the property does not hold. Such error traces are useful to modellers, since they may provide insights into the root causes of unexpected behaviour.
- It is possible to check partial specifications for system correctness, which makes it possible to avoid the modelling of complete systems, if this is desirable in an evaluation scenario.

However, model checking suffers from the same problem as all Markovian systems: state space explosion. This problem occurs when the model has many components that perform transitions in parallel. In such a case, the number of states of the model usually grows exponentially with the number of components. Hence, the main challenge in model checking is the tackling of the state space explosion problem. The process of model checking a system is shown in Figure 2.13, and can be summarised as follows:

1. *System Modelling:* A system firstly needs to be represented mathematically in the form of a model.
2. *Property Specification:* Before verification can commence, the model checker has to be informed of the properties of interest that are to be verified. They are generally

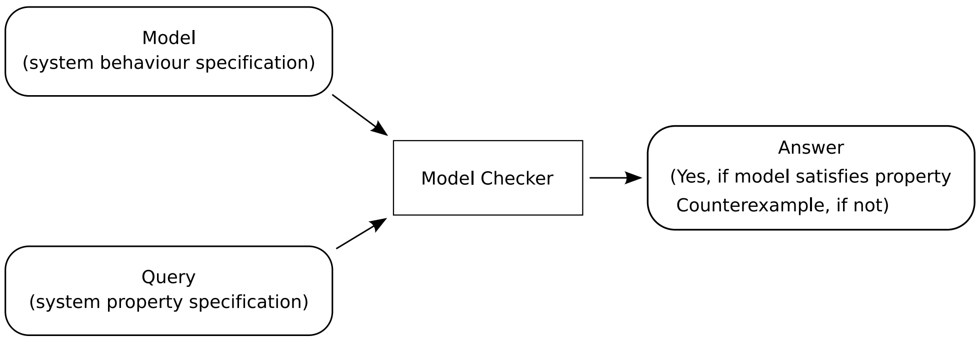

Model
(system behaviour specification)
Query
(system property specification)
Model Checker
Answer
(Yes, if model satisfies property
Counterexample, if not)

Figure 2.13: The process of model checking

specified in some logical formalism. For hardware and software systems, temporal logics are used, which are able to express the behavioural evolution of systems over time.

3. *Verification:* The verification process is automatic; however, it does involve some degree of human interaction, mostly in the form of analysing verification results. Formally, verification can be stated as follows: given a particular property, expressed as a temporal logic formula p, and a model M with initial state s, decide if $M, s \models p$.

While model checking is a powerful tool for designers and engineers wishing to verify their systems, it is not the method of choice for purely quantitative performance analysis, since model checkers are unable to provide direct quantitative results that relate to system performance. Realising this, the model checking community has attempted to diversify the process to also support a limited form of quantitative analysis by investigating qualitative and quantitative model checking algorithms for probabilistic systems. In a qualitative setting, the aim is to check whether a property holds with probability 0 or 1, whereas in a quantitative setting, it is to be verified whether the probability of a certain property being satisfied meets given lower or upper bounds, which are different from 0 and 1.

Much research has been carried out on verification methods for probabilistic logics. Probabilistic extensions of modal and temporal logics and automatic procedures for verifying the satisfaction for such logics have been developed. These are mainly based on reducing the calculation of the probability of formulae being satisfied to a linear algebra problem.

2.3.2 Numerical Analysis

Steady-State Analysis

Performance analysis is often concerned with the behaviour of systems over an extended period of time. Hence, the primary objective of a modeller is the calculation of the probability distribution of a model of a system over the state space $\mathcal{S}$ as it settles into a regular pattern of behaviour. Intuitively, this means that the system has been running for a long time and its behaviour no longer exhibits any trends. This probability distribution is commonly referred to as the *steady-state distribution* and is very useful for deriving performance measures based on subsets of states where some conditions hold.

Since we may have to choose an initial state for a model randomly, by considering the long-term probability distribution we can balance out any form of bias that could possibly have been introduced by the chosen start state. Under certain conditions, the more steps the system takes, the less it matters what state it was in at the time when it started. We assume that when the effects of initial bias have worn off, the system is in *steady state*. This does not imply that it is stuck in a particular state and no longer evolves; rather that it is assumed to exhibit regularity and predictability in its behaviour. It continues to change state, but the probability of observing it in any given state is no longer a function of time. This is reflected by the absence of change in the probability distribution. Systems that are able to reach steady state are said to be stable [Hillston04, Mitrani98].

Let $\pi_j(t)$ denote the probability that an irreducible, aperiodic and time-homogeneous Markov process $\{\mathcal{X}(t)\}$ is in state j at time t. In the limit, when the observation instant is infinitely far removed from the starting point, the probability of finding the system in state j is independent of the initial state. Steady state has been reached at time t when for all states j and all $\tau > 0$ we have that $\pi_j(t+\tau) = \pi_j(t)$, i.e. the time at which the system is observed does not influence the probability of it being in a particular state. Therefore, we denote steady-state probabilities without the time variable by π_j:

$$\pi_j \ = \ \lim_{t\to\infty} \mathbb{P}(\mathcal{X}(t) \ = \ j \mid \mathcal{X}(0) = i) \tag{2.26}$$

for $i, j = 0, 1, \ldots$. When the limiting probabilities π_j exist, they represent the steady-state distribution of the Markov process. A steady-state distribution, $\{\pi_j : j \in \mathcal{S}\}$, exists for every time-homogeneous, finite and irreducible Markov process [Hillston04].

In steady state, π_j is the proportion of time that the process spends in state j. Hence, at a moment in time, the probability of a transition occurring that moves the system from state i to state j is given by the probability that the model is in state i, multiplied by the instantaneous probability of the system making a transition from state i to state j, $\pi_i q_{ij}$, which is also called the *probability flux* from state i to state j. In order for equilibrium to be maintained, the total probability flux into a state has to be equal to the total probability flux out of the state. Otherwise, the probability distribution over the set of states would change. Hence, for any state i, we have:

$$\sum_j \pi_j q_{ji} = \pi_i \sum_j q_{ij} \tag{2.27}$$

The left hand side of Equation 2.27 represents the flux into state i, while the right hand side represents the flux out of it. Equations of this form for all states $i \in \mathcal{S}$ are collectively called *global balance equations*. Since the the sum of elements in every row of the generator matrix $\boldsymbol{Q}$ is zero, i.e. $\sum\limits_j q_{ij} = 0$, Equation 2.27 can also be written as:

$$\sum_j \pi_j q_{ji} = 0 \tag{2.28}$$

Together, the π_j values represent the steady-state probability distribution and are not known in advance. If there are n states in the state space, n such equations need to be solved to find the n unknowns. However, due to redundancy inherent in the equations, not enough information is available to solve them uniquely. Forming a row vector out of the π_i values, we can express Equation 2.28 in matrix equation form as:

$$\boldsymbol{\pi Q} = \boldsymbol{0} \tag{2.29}$$

Since $\{\pi_j\}$ represents a probability distribution, we also know that the normalisation condition applies:

$$\sum_{x_j} \pi_j \;=\; 1 \tag{2.30}$$

Together with Equation 2.30, we now have $n+1$ equations for n unknowns, which enables us to obtain a unique solution. When the state space of an irreducible Markov process is finite, the process is always recurrent non-null, and therefore, Equations 2.29 and 2.30 always have a solution [Ross82].

There are mainly two types of solution methods for calculating the steady-state distribution. *Direct methods* obtain an exact solution after a finite number of steps, whereas *iterative methods* produce approximate solutions. Direct methods are generally appropriate when the state space of the model is not particularly large and when the corresponding state transition matrix is not sparse. Direct methods also have the advantage that they impose an upper bound on the time taken to obtain a solution. In contrast, iterative methods are appropriate when the state transition matrix is large and sparse, since the methods preserve the sparsity of the matrix. Iterative methods require less storage and computational resources, but at the same time often require a long time to converge towards a solution.

From the steady-state solution, several measures of interest can be calculated. *State-based measures* are those that correspond to the probability of a system being in a particular state, or a set of states, that satisfy some condition. Good examples of state-based measures are utilisation or the distribution of the number of customers in a system. *Rate-based measures* are those that correspond to the predicted rate at which events occur. An example of such a measure is throughput.

Transient Analysis

Another important class of performance analysis is *transient analysis*, which aims to find the probability of a system being in a certain state at time t.

Steady-state probabilities refer to system behaviour in the long run, while transient probabilities consider the system at a fixed time instant. Transient analysis is more meaningful than steady-state analysis when systems need to be evaluated with respect to their short-term behaviour. In the analysis of systems that have to remain operational during a certain period of time, such as on-board navigational computers found on airplanes or satellite control systems, for example, possible questions may relate to the probability of systems failing at some point. In scenarios like these, modellers have to calculate measures within a relatively short time interval. Results obtained from the steady-state solution, which characterise system behaviour in the long run, are not useful approximations for the desired measures and could lead to substantial errors. Measures that can be derived from transient state probabilities are often referred to as *instantaneous measures*.

The process of *uniformisation* has classically been used to conduct transient analyses of CTMCs [Jensen53, Grassman87, Reibman88]. The transient state distribution of a CTMC is defined as the probability of the process being in one of the set of states J at time t, given that it has started in state i at time 0:

$$\pi_{iJ}(t) \ = \ \mathbb{P}(\chi(t) \in J \mid \chi(0) = i) \tag{2.31}$$

where $\chi(t)$ denotes the state of the CTMC at time t. Transient uniformisation takes advantage of the fact that for any given ergodic CTMC a corresponding DTMC can be constructed, which yields a steady-state probability vector that is identical to that of the CTMC. In a uniformised CTMC, the probability that the CTMC is in a state in J at time t is calculated by conditioning on $N(t)$, the number of transitions that occur in a given time interval $[0, t]$ in a DTMC [Bolch98, Bradley06]:

$$\pi_{iJ}(t) \ = \ \sum_{m=0}^{\infty} \mathbb{P}(\chi(t) \in J \mid N(t) = m)\mathbb{P}(N(t) = m) \tag{2.32}$$

Other approaches for the calculation of transient measures exist, such as Laplace transform–based methods (see [Dingle04a] for details), but we do not consider them further here.

Passage Time Analysis

Passage (or *response*) *times* are important QoS metrics of stochastic systems that describe the time that systems take to enter for the first time one of a set of target states J, given that they have started in one of a set of start states I.

Passage time analysis empowers modellers to ask a wide range of relevant and informative performance questions that address the evolution of systems between two distinct moments in time. Some of the most important areas of its application are reliability testing, efficiency analysis and the verification of conformity to SLAs. During reliability testing, system engineers query models to obtain probability measures representing the likelihood of their systems failing within certain periods of time by entering an error state. Efficiency analysis addresses the amount of time that a system takes to perform a particular task, and considers these times an indicator of system responsiveness. In Markovian models, passage times are mainly calculated using two distinct approaches. The first is based on uniformisation, and the second on a Laplace transform method.

Uniformisation-based Analysis

Uniformisation has classically been applied to the transient analysis of CTMCs, but it can also be used for the calculation of passage time densities, as described in [Melamed84, Muppala92, Bolch98, Miner03]. It transforms a CTMC's states to have the same mean holding time by allowing invisible transitions from states to themselves. The interval between a moment where a CTMC enters state i, and the first subsequent moment when it enters one of the states in J is called the *first passage time* from i to J and is defined as:

$$P_{iJ} \;=\; \inf\{t > 0 \,:\, \chi(t) \in J, N(t) > 0, \chi(0) = i\} \tag{2.33}$$

where $N(t)$ denotes the number of state transitions that have occurred by time t. Note that this approach only calculates the first passage time density, and does not consider repeated visits to target states. The density of the passage time between states i and j in the uniformised chain can be expressed as the sum of m n-stage Erlang state holding time densities, weighted by the probability of the CTMC moving from state i to state j in exactly n hops, such that $1 \leq n \leq m$. This result can also be generalised for multiple start states, by additionally providing a probability distribution across them. The pdf of the response time between the non-empty set of source states I and the non-empty set of target states J in the uniformised chain is given by [Bradley06]:

$$f_{IJ}(t) = \sum_{n=1}^{m} \left(\frac{q^n t^{n-1} e^{-qt}}{(n-1)!} \sum_{k \in J} \pi_k^{(n)} \right) \tag{2.34}$$

where

$$\boldsymbol{\pi}^{(n+1)} = \boldsymbol{\pi}^{(n)} \boldsymbol{P'} \tag{2.35}$$

with

$$\pi_k^{(0)} = \begin{cases} \dfrac{\pi_k}{\sum_{j \in I} \pi_j} & \text{if } k \in I \\ 0 & \text{otherwise} \end{cases} \tag{2.36}$$

$\boldsymbol{P'}$ is a modified transition probability matrix with all target states made absorbing. The π_k values are the steady-state probabilities of the corresponding state k from the CTMC's embedded Markov chain.

Laplace Transform-based Analysis

Similarly to the uniformisation-based approach, the first step in the Laplace transform-based method is the calculation of the first passage time density of the system that has started in state i entering any state out of the set of target states J. Hence, it is necessary to calculate the convolution of state holding time densities over all possible paths from state i to J. This approach takes advantage of the favourable properties of the *Laplace transform*. For a given real-valued function $f(t), t \geq 0$, the Laplace transform, denoted by $L\{f(t)\}, f^*(s)$ or $L(s)$, is defined as [Nelson95]:

$$L\{f(t)\} \;=\; f^*(s) \;=\; L(s) \;=\; \int_0^\infty e^{-st} f(t) dt \tag{2.37}$$

In essence, the Laplace transform converts a function from the real-valued time domain (t space) to the complex-valued Laplace domain (s-space). Due to its algebraic properties, operations that are complex in t-space become simple in s-space. The following properties of Laplace transforms are particularly useful:

- The differentiation of a function in t-space corresponds to the multiplication of the

Laplace transform of the function by s in s-space:

$$L\{f'(t)\} = sf^*(s) \tag{2.38}$$

- The integration of a function in t-space corresponds to the division of the Laplace transform of the function by s in s-space:

$$L\{\int_0^t f(\tau)d\tau\} = \frac{f^*(s)}{s} \tag{2.39}$$

- The convolution of two functions in t-space corresponds to the product of their individual Laplace transforms in s-space:

$$L\{(f \circ g)(t)\} = f^*(s) * g^*(s) \tag{2.40}$$

- The n^{th} moment of a probability density function $f(t)$ of the continuous random variable χ in t-space can be obtained in s-space by calculating:

$$E(\chi^n) = (-1)^n f^{*(n)}(0) \tag{2.41}$$

After carrying out Laplace transform-based calculations, results need to be numerically inverted, in order to convert them back into t-space. The calculation of the passage time density function is therefore achieved by calculating the Laplace transform of the convolution of the state holding time densities over all paths between i and J, and then numerically inverting the resulting function [Harrison02]:

$$L_{iJ}(s) = \sum_{k \notin J} \left(\frac{q_{ik}}{s - q_{ii}}\right) L_{kJ}(s) + \sum_{k \in J} \left(\frac{q_{ik}}{s - q_{ii}}\right) \tag{2.42}$$

When there are multiple source states, denoted by I, the Laplace transform of the passage time density at steady state is [Bradley03d]:

$$L_{IJ}(s) = \sum_{k \in I} \alpha_k L_{kJ}(s) \tag{2.43}$$

where the weight α_k is the probability at equilibrium that the system is in state $k \in I$ at the starting instant of the passage, and is given by:

$$\alpha_k = \begin{cases} \dfrac{\pi_k}{\sum\limits_{j \in I} \pi_j} & \text{if } k \in I \\ 0 & \text{otherwise} \end{cases} \tag{2.44}$$

Comparison of Approaches

Uniformisation is generally much faster than the Laplace transform-based method, except in the case of very small models. However, the Laplace transform method is easier to extend to semi-Markov systems with generally distributed state holding times, and it preserves the ability to reason about vanishing source and target states. Uniformisation is unable to support the latter kind of reasoning, since vanishing states are assumed to be eliminated during state space generation.

2.3.3 Simulation

The simulation approach of analysing a model is an alternative to the analytical approach, where system analysis is purely theoretical. Stochastic models, which are solved analytically for performance measures, are mathematical abstractions of systems. In contrast, stochastic simulation models can be regarded as algorithmic abstractions, which reproduce the behaviour of systems that they represent when executed. Modelling complex systems theoretically requires many simplifications, and emerging models are often not accurate representations of reality.

Simulation, on the other hand, does not require as many simplifying assumptions, which eliminates this problem. As simulation models are run rather than solved, performance measures are observed rather than derived. However, a single observation is generally not sufficient, and multiple simulation runs are often required for results to be conclusive. Simulation models may in certain circumstances offer greater freedom in the modelling of important aspects of system behaviour than other approaches, and they also enable models to be considered whose state space exceeds analytical tractability.

Simulation models allow the modeller in theory to represent systems at arbitrary levels of detail; however, in practice there is a trade-off between the realism of the model and the time that it takes to produce a statistically significant run. In simulation models, the state space is generated during execution by the models themselves, which eliminates the need to store it all at once, as is often required by the analytical approach. At the same time, simulation models can sometimes be very time-consuming to create, since the specification process involves writing and debugging potentially complex code. Simulation models are also expensive to evaluate, because simulation runs require substantial computational resources, and a number of them are usually necessary in order to obtain relevant metrics [Hillston04].

Simulation models are complex computer programs, which can be developed in any programming language. Most often, a system model is constructed either in the form of a computer program or as some kind of input to simulator software. The process of simulation usually takes place in the following order:

1. *Problem definition:* Inputs and constraints on decision variables are identified; then, the measure of system performance is defined, followed by the development of a preliminary model structure that relates inputs and the measure of performance.

2. *Data collection and analysis:* A method used for the collection of data is defined, taking into account the fact that regardless of the chosen method, the decision of how much data to collect effects a trade-off between cost and accuracy.

3. *Development of simulation model:* Sufficient knowledge of a system needs to be acquired in order to develop an appropriate conceptual and logical model. This is then followed by a more detailed simulation model.

4. *Model validation, verification and calibration:* Validation ensures that models correspond to reality, and verification establishes whether their implementation corresponds to the conceptual model. Thus, validation addresses the question *“Is the right system being built?”*, whereas verification is concerned with the question *“Is*

the system being built right?" Finally, calibration ensures that data generated by the simulation matches actual observed data.

5. *Input and output analysis:* Discrete event simulation models typically have stochastic components that replicate the probabilistic behaviour of systems. Accurate input modelling requires a close match between an input model and the underlying probabilistic mechanism of a system. Input data analysis models an element (e.g. an arrival process or service times) in a discrete event simulation, given a data set that was collected on the element of interest. Error checking is performed on input data, and simulation experiments are carried out to derive conclusions from simulated system behaviour.

6. *Sensitivity estimation:* Provides the means for modellers to understand which relationships are meaningful in complicated models.

7. *Reporting:* Ensures the provision of relevant simulation results to modellers.

2.3.4 Comparison of Techniques

Probabilistic Model Checking

The main purpose of probabilistic model checking is the verification of behavioural properties of system models. Systems are modelled as Markovian processes, which are theoretical abstractions that are solved in order to obtain performance results. Models are generally solved with numerical analysis techniques, according to criteria defined in performance queries. Queries are expressed in formulae of stochastic logics, and are able to reason about state- and path-based constraints, as well as steady-state measures. Probabilistic model checking is interested in obtaining *yes / no* answers, which indicate whether or not certain properties defined in performance queries are satisfied by the model. Specialised model checkers support and automate the verification process. One of the main limitations of probabilistic model checking is that very large models cannot be verified, due to the state space explosion problem, and that quantitative performance measures cannot be extracted from models.

Numerical Analysis

Numerical analysis applies mathematical solution techniques to Markovian and semi-Markovian models in order to derive performance measures of interest. Similarly to probabilistic model checking, systems are modelled and subsequently solved. Unlike probabilistic model checking, numerical analysis is not restricted to verification-style analyses, but is able to extract a wide range of performance metrics from models. Some assumptions about systems are required, especially with respect to the timing of events, but the resulting models are relatively easy to solve, as they only rely on simple linear algebra techniques. A number of dedicated analysis tools exist that implement numerical solution techniques, and they can be used in many diverse application scenarios. On the downside, complex models with very large state spaces can often not be analysed by currently available numerical algorithms due to the state space explosion problem that imposes computational resource requirements that exceed availability.

Simulation

Simulation differs from the previous two approaches mainly in that it abstracts systems algorithmically, and observes results of runs to obtain estimates. Simulation models are generally speaking less sensitive to the size of the state space and allow for less simplified models to be analysed than other approaches, which makes simulation models widely applicable and very powerful. However, their design and execution can be time-consuming, and the evaluation of the trustworthiness of results is required through the calculation of confidence intervals.

2.4 Tool Support for Performance Analysis

2.4.1 Tools for Performance Analysis

Below, we present some of the more well-known tools for performance analysis. Some of the tools are exclusively model checkers or quantitative analysers, while others are equipped for a more versatile application by featuring support for both model checking and quantitative analysis.

GreatSPN

GreatSPN, the Graphical Editor and Analyser for Timed and Stochastic Petri Nets [Chiola95], is a software package for the modelling, validation and performance evaluation of distributed systems, represented by GSPNs and their coloured extension, stochastic well-formed nets. The tool provides a framework for timed Petri net-based modelling, and implements efficient algorithms to enable the analysis of complex large-scale systems. *GreatSPN* consists of a number of separate tools that collaborate in the construction and analysis of Petri net models. Different analysis modules can be run on different machines in a distributed environment, and the modular structure of the tool makes it receptive to the addition of new analysis modules. *GreatSPN*'s main features are:

- The *graphical user interface* (see Figure 2.14) enables the graphical editing of Petri net models and the representation of structural properties. It enables the definition of timing and stochastic specifications, parameters and performance measures, provides menu-driven interaction with individual analysis modules, and presents performance results in a graphical fashion. In addition, it features an interactive simulation and token game for Petri nets with priorities and inhibitor arcs.
- It allows *structural properties* for nets with priorities and inhibitor arcs to be checked.
- It uses *linear programming* for the calculation of performance bounds of GSPNs.
- Its integrated *Markovian solvers* for steady-state and transient performance evaluation exploit efficient sparse matrix-based numerical techniques.
- It features *simulation modules* for interactive event-driven simulation. In cooperation with the GUI, they provide graphical model animation, real-time updating of performance measures and arbitrary rescheduling of events.
- The *well-formed coloured net module* supports the construction of coloured and

symbolic reachability graphs, and their conversion into lumped Markov chains. The module supports steady-state and transient analysis, as well as simulation.

- *Support for CSL model checking on GSPN models* is realised by interfaces to *PRISM* and *ETMCC* [D'Aprile04]. Interfacing with *PRISM* is realised by a translation of GSPNs into the state-based *PRISM* input language on the net level. CSL formulae that express performance requirements are specified in *PRISM*'s graphical interface. When interfacing with *ETMCC*, model checking is realised by the translation of GSPN models to CTMCs. A translator creates a CTMC from the GSPN model in the format that is expected by *ETMCC*, and users specify properties in *ETMCC*.

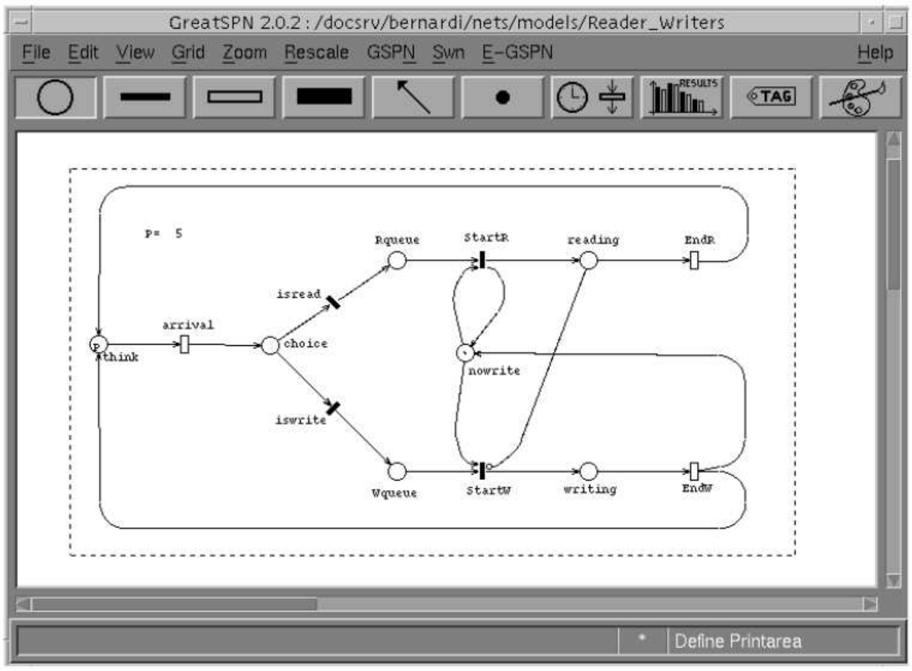


Figure 2.14: *GreatSPN* user interface

DNAmaca

DNAmaca [Knottenbelt96] is a Markov chain steady-state analyser that is able to solve models with up to $O(10^8)$ states. It features model and performance measure specification in its input language, and provides support for the complete performance analysis sequence by enabling model specification, state space generation, functional and steady state analysis and the computation of performance measures. The tool consists of a number of components, whose interaction is also illustrated in Figure 2.15:

- The *parser module* translates high-level model descriptions into C++ classes.
- A *state space generator* is formed for each model by linking the corresponding C++ class with common library routines. The state space generator uses a probabilistic exploration algorithm, incorporating vanishing state elimination, to generate all reachable tangible states. The infinitesimal generator matrix, describing transition rates between tangible states, is also generated.
- The *functional analyser* checks the generator matrix for Markov chain irreducibility, which is a necessary precondition for a stationary distribution solution.
- The *steady-state solver* determines the stationary distribution by solving the set of global balance equations for the model.
- The *performance analyser* is formed by linking user code with common library routines. It uses the steady-state solution in combination with state space information to calculate performance measures. *DNAmaca* is able to calculate state and count measures. A state measure is used to determine the mean and variance of a real expression that is defined at every state of a system. A count measure is used to determine the mean rate at which a particular event occurs.

HYDRA

HYDRA [Dingle04a] facilitates the parallel and distributed analysis of very large Markov models for passage time and transient state measures through the use of uniformisation [Grassman87, Reibman88]. *HYDRA* builds on *DNAmaca* technology, and in addition supports parallelised performance measure computation using state-of-the-art hypergraph partitioning algorithms [Trifunović04], which enable the efficient distribution of sparse matrix-vector operations across a number of processors.

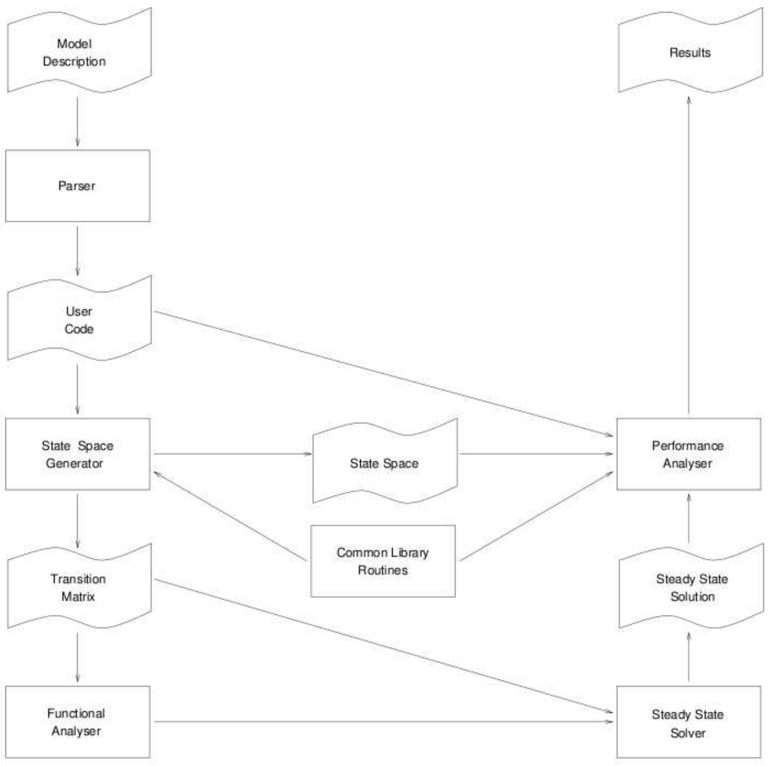


Figure 2.15: *DNAmaca* module interaction

Figure 2.16 shows *HYDRA*'s architecture. At the beginning of the analysis process, a high-level model is specified in *DNAmaca*'s model specification language. Following that, *HYDRA*'s state generator produces the generator matrix of the underlying Markov chain, together with a list of start and target states in the case of a passage time analysis run. Uniformisation is then applied to transform the generator matrix, which is subsequently partitioned using hypergraph algorithms. Distributed passage time and transient analysis modules are then invoked to calculate desired performance metrics.

SMARTA

SMARTA [Dingle04a] is a parallel and distributed MPI-based semi-Markov response time analyser that incorporates *DNAmaca* technology. It performs iterative numerical analyses of passage times in very large semi-Markov models (including GSPNs), using Laplace transform inversion and hypergraph partitioning techniques.

SMARTA's tool architecture is shown in Figure 2.17. The passage time analysis process

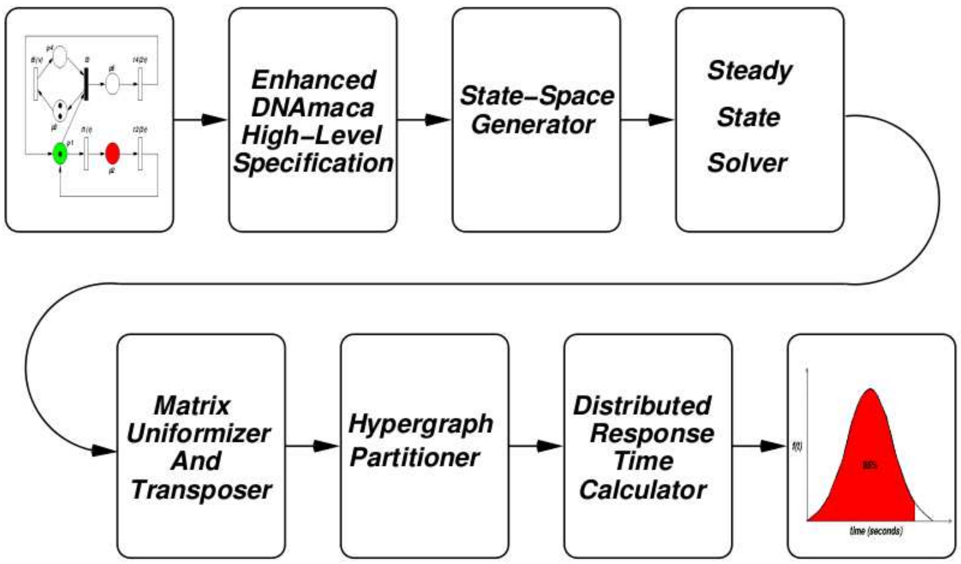

Enhanced DNAmaca High–Level Specification
State–Space Generator
Steady State Solver
Matrix Uniformizer And Transposer
Hypergraph Partitioner
Distributed Response Time Calculator
time (seconds)

Figure 2.16: *HYDRA* tool architecture

is similar to that of *HYDRA*. Passage time results are provided in the form of text files, which can be parsed by GNUplot for graph visualisation.

SHARPE

SHARPE, the Symbolic Hierarchical Automated Reliability and Performance Evaluator [Hirel00], supports the construction and analysis of performance, reliability, availability and performability models. Among others, *SHARPE* supports Markov and semi-Markov chains and GSPNs, and large models can be constructed elegantly by using hierarchical model composition. It also provides flexible mechanisms for combining results, so that models can be used in hierarchical combinations.

SHARPE has a graphical user interface, whose major components are a model editor that allows graphical model definitions, and an extensive collection of visualisation routines to analyse output results. The interface also provides a high-level input format to the *SHARPE* syntax. Users are able to create Markov chain-based models by defining underlying probability matrices. The GUI provides a way to plot results, and supports the exporting of data into Excel spreadsheets. The interface was written in Java to make the tool architecture-independent and portable.

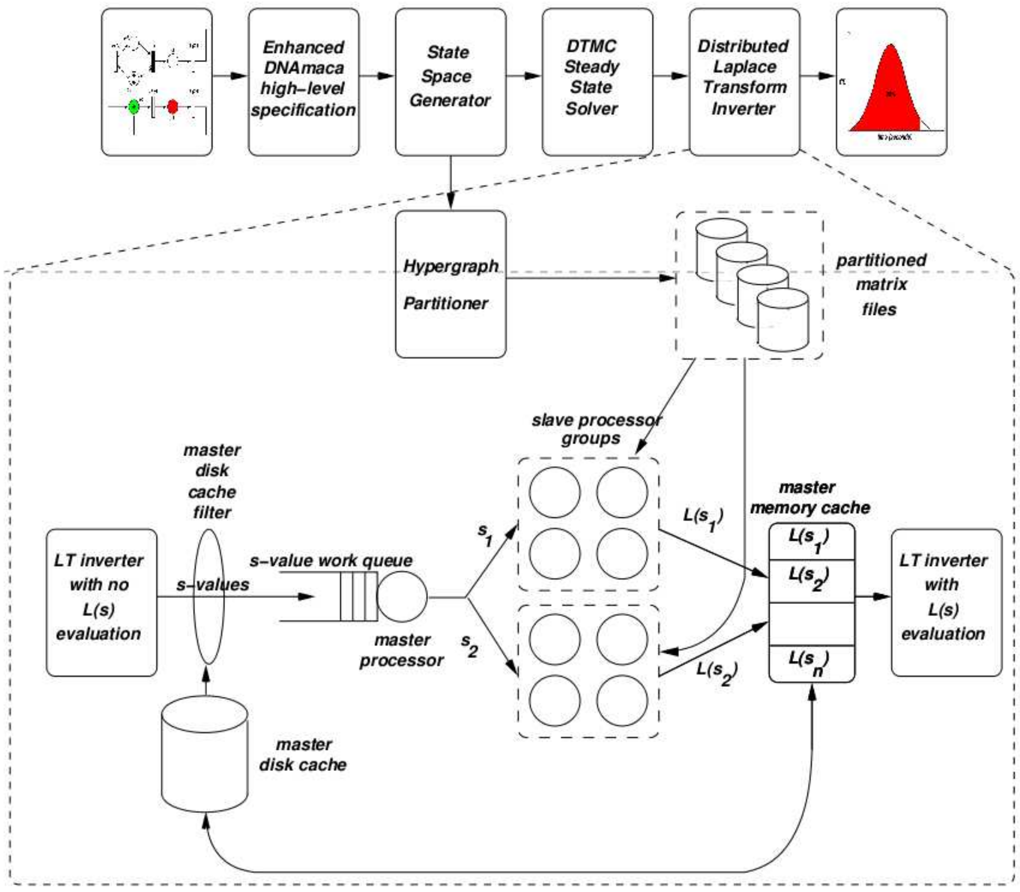


Figure 2.17: *SMARTA* tool architecture

Möbius

Möbius [Clark01] is a tool for modelling the behaviour of complex computer and network systems and for studying their reliability, availability and performance. Its fundamental assumption is that no modelling formalism can ever be the single best approach to constructing all different kinds of system models. Many domain-specific modelling languages and techniques for analysing models, such as simulation, state space exploration and analytical solution, are needed to study important system behaviour. Therefore, *Möbius* defines a broad framework into which new modelling formalisms and model solution methods can be integrated to collaborate with the set of already supported formalisms and techniques. This flexibility allows engineers and scientists to represent systems with modelling languages that are appropriate to their specific problem domains, and accurately and efficiently solve systems using solution techniques that are best suited to their size and complexity. Time- and space-efficient discrete event simulation and numerical solution, operating on Markov processes, are both supported. *Möbius*' main features are:

- *Support for multiple graphical and textual modelling languages*. Model types include SPNs, CTMCs with extensions and SPAs. Models are constructed with the appropriate level of detail and customised to the specific behaviour of the system of interest.
- *Ability to define customised system measures* in the form of detailed expressions, based on nodes and activities in models. Such measures can relate to reliability, availability, performance and security. Measurements can be conducted at specific time points, over periods of time, or when systems have reached steady state.
- *Study of system behaviour under a variety of operating conditions* is possible, since system functionality can be defined by model parameters. These can be observed across a wide range of values in order to study system behaviour that could be challenging to measure with prototypes.
- *Distributed discrete event simulation* enables the evaluation of custom measures using efficient simulation algorithms. Systems are executed repeatedly as simulation runs, either locally or across remote clusters in a distributed manner, and statistical results of measures are gathered.
- *Numerical solution techniques* are used for obtaining exact solutions for many classes of models. Advances in state space computation and generation techniques enable the solution of models with tens of millions of states, which could previously only be solved by simulation.

PRISM

PRISM [Kwiatkowska02] is a tool for the formal modelling and analysis of probabilistic systems, defined by DTMCs, CTMCs, Markov decision processes, their cost- and reward-based extensions, PEPA and SBML, the Systems Biology Markup Language. It enables the automated analysis of a wide range of quantitative properties of such models. Symbolic data structures and algorithms, based on Binary Decision Diagrams (BDDs) and

Multi-Terminal Binary Decision Diagrams (MTBDDs) are used for efficient model representation, enabling the analysis of large-scale models. In addition, *PRISM* also supports discrete event simulation for generating approximate quantitative results.

Models are described using the state-based *PRISM* language, a probabilistic variant of the Reactive Modules language. Its fundamental components are modules and variables, and a model is composed of a number of modules that can interact with each other. A module contains a number of local variables, and the values of these variables at any given time constitute the state of the module. The global state of the model is determined by the local state of all modules. The behaviour of each module is described by a set of commands, which take the following form:

```
[] guard -> prob_1 : update_1 + ... + prob_n : update_n;
```

A guard is a predicate over all variables in the model. Updates describe transitions that the module can make if the guard conditions are satisfied, and transitions are performed by updating variables in the affected module. Each update is also associated with a probability or rate which is assigned to the corresponding transition. The following example shows the description of a module:

```
module M1
    x : [0..2] init 0;
    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);
endmodule
```

PRISM's property specification language (see Section 2.2.4) allows the expression of PCTL and CSL formulae, as well as cost- and reward-based quantitative properties. The tool takes as input a description of a system, from which it constructs a model and computes the set of reachable states, and a property specification file that defines which properties need to be verified on the model. Model construction and reachability calculation are implemented with space-efficient symbolic representations. Model checking is carried out using a combination of reachability-based computation and the solution of linear equation systems.

PRISM has been extended to interface with Grid clusters and carry out computations on them [Zhang05]. After the calculation of the probability matrix and initial vector for a model, they are exported to files and transferred to remote systems. Following this, the job submission component submits a model checking job to the remote system. Users are able to monitor job progress through a dedicated monitoring component. Once a job completes remotely, the result vector is transferred back. *PRISM* uses the Globus Toolkit as the basis of its Grid middleware, which implements file transfer, job management and monitoring. The Midlands eScience Cluster [MeSC] serves as *PRISM*'s underlying Grid computational infrastructure. Figure 2.18 shows *PRISM*'s enhanced Grid-enabled architecture.

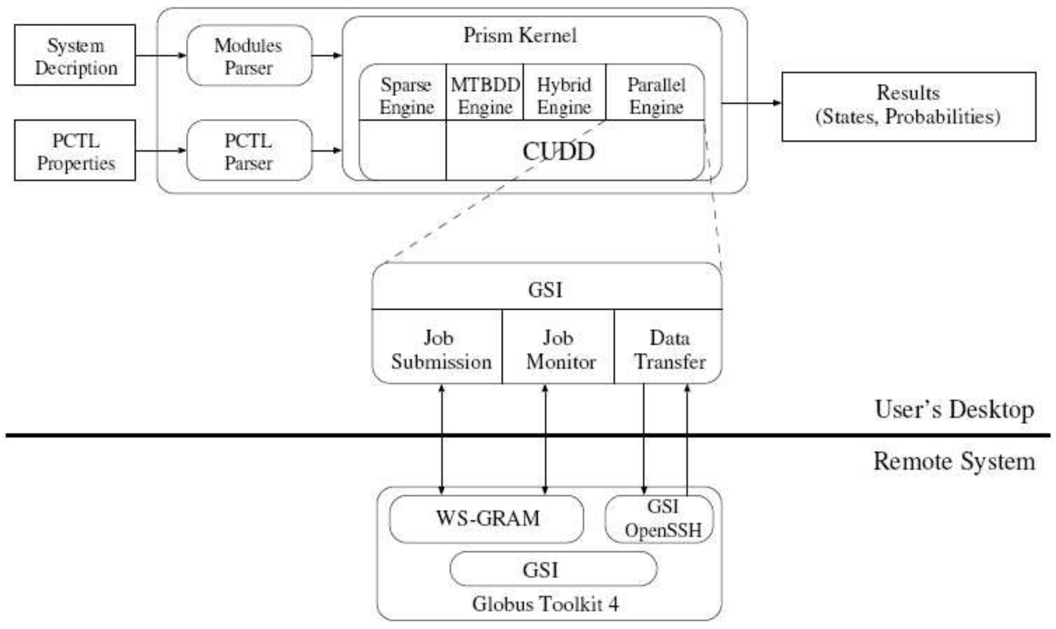


Figure 2.18: Grid-enabled *PRISM* tool architecture

PRISM's high-level modules, such as the GUI and parser, are written in Java, while low-level code and libraries were implemented in C++. The tool can be run with a GUI or from the command line.

Chapter 3

Performance Trees

Systems engineers are faced with high expectations to design and build systems that meet end-user operational performance requirements. This is a particularly challenging task for large-scale, high-throughput distributed systems, such as cluster computers and telecommunication networks for example. An established pipeline for determining whether a given system meets its expected performance is to

(a) construct a mathematical model that replicates its behaviour, using some stochastic modelling formalism (as presented in Section 2.1),

(b) express applicable performance-related queries in terms of requirements and measures, using a stochastic logic or other methodology (see Section 2.2), and

(c) apply specialised stochastic model checking or quantitative analysis software (see Section 2.4) to resolve queries.

This chapter addresses step (b) of the process by introducing Performance Trees, a graphical formalism for the specification of performance requirement- and quantitative measure-based queries on stochastic system models.

We begin by presenting the motivations for the development of the formalism and give a brief overview of its main features. We then discuss the structure of Performance Tree queries, and introduce the set of currently available operators. Once the fundamentals have been established, we highlight the power of Performance Trees by elaborating on their accessibility, expressiveness, extensibility and versatility. Finally, we provide examples of Performance Tree query specification to demonstrate the use and applicability of the formalism.

3.1 A Novel Representation Formalism for Performance Queries

3.1.1 Motivations

To study the behaviour and observe various interesting properties of real-life systems, engineers construct stochastic models that represent systems mathematically, in a manner that is amenable to analysis. To analyse system models, engineers need to construct performance queries that specify performance properties and/or measures of interest.

As Section 2.2 has shown, a number of different approaches can be taken for the specification of performance queries. For a number of reasons, stochastic logics have been among the most widely-used methods for performance requirement specification. Due to their logical nature, they have well-defined syntactic structures and semantics, and their formulae are able to express performance properties in a concise, rigorous and systematically verifiable manner. However, their use among system designers is rather limited in practice, due to their inherent complexity and restricted expressive power. The conciseness and nested nature of stochastic logic formulae has a tendency to obscure questions being asked. In addition, an expert understanding is often required to translate performance requirements expressed in natural language into logical formulae. For these reasons, stochastic logics are perceived as esoteric by many industrial users and have not gained wide acceptance in such circles.

Modern Service Level Agreements (SLAs) include increasingly complex performance properties. Hence, system designers need to establish already at design-time whether their systems are going to meet QoS requirements set out by SLAs. With current approaches to performance query specification, such as stochastic logics for example, many performance properties of interest cannot be addressed due to limitations in expressiveness. Such properties often relate to quantitative performance measures that need to be extracted from system models directly.

Traditionally, it has only been possible to reason about performance measures with tool-oriented and graphical query specification languages. Tool languages were developed for individual quantitative analysers, and are therefore very specific and limited in terms of their expressiveness and application. Generally, they are not exclusively query specification languages, but ones that also incorporate model specification capabilities. The same applies to most graphical languages, which generally only annotate system specifications with performance information.

In summary, no single formalism has been available so far that would enable the concise and accessible expression of performance requirements *and* performance measures in a single query. As such, the combined expressive power of current approaches to performance query specification is not sufficient for many of the more sophisticated analysis scenarios.

3.1.2 Overview

To cater for these requirements and to overcome the aforementioned shortcomings of current approaches to query specification, we have developed *Performance Trees* [Suto06a, Suto06b, Suto07], a graphical formalism for compositional performance query specification that enables the reasoning about performance requirements and the direct extraction of quantitative performance measures. Performance Trees support elegant query composition, are easily visualised as hierarchical tree structures, and provide a general framework that allows for the expression of a wide range of performance properties in a uniform manner. In addition, they are applicable in the context of several modelling formalisms, including SPNs, QNs and SPAs, due to the use of an abstract state specification mechanism. Moreover, Performance Trees are extensible, either by a parameterised macro mechanism that uses existing operators to construct user-defined performance concepts, or by the incorporation of new operators. Hence, Performance Trees represent an accessible, powerful and versatile alternative for performance query specification.

Performance Tree queries are constructed from a rich selection of operators, visualised graphically as tree nodes, which represent performance concepts and properties that are familiar to users with an engineering background. The core set of operators can be extended dynamically to include support for additional user-defined concepts.

Currently, Performance Trees are able to reason about steady-state and passage time properties by specifying applicable probability distributions and densities. Queries are able to address properties that are derived from these distributions and densities, such as moments, percentiles and convolutions. In addition, probabilities with which passages take place in a given amount of time and the transient probability of systems being in a set of states at a given point in time can also be expressed. Performance Trees allow the reasoning about mean rates of occurrence of system actions, about states that have certain steady state probabilities, and about states that a system can occupy at a given point in time with a certain probability. In addition to performance-related concepts, it is also possible to specify Boolean and arithmetic operations and comparisons, as well as macros, which condense possibly very complex user-defined performance properties into single Performance Tree nodes.

The power of Performance Trees is provided without sacrificing computational tractability, since all operators either impose a trivial computational burden or are backed up by known numerical algorithms that are amenable to scalable parallel implementation.

3.1.3 Query Specification with Performance Trees

While Performance Tree queries can also be expressed in a textual form, the formalism was primarily designed for graphical user-level specification. A visualised instance of a Performance Tree query consists of a set of nodes that form a hierarchical tree structure when connected by arcs, as shown in Figure 3.1.

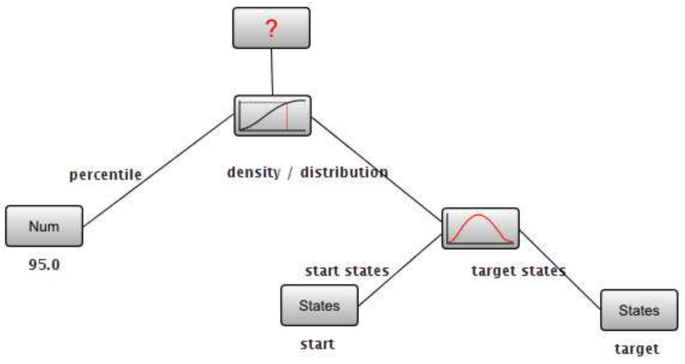


Figure 3.1: An example Performance Tree query

Nodes in a Performance Tree query can be of two kinds: operation or value nodes. *Operation nodes* represent performance concepts and behave like functions, taking one or more child nodes as arguments and returning a result. Child nodes can be other operation nodes that return a value of an appropriate type or *value nodes*, which usually represent states, functions on states, actions, numerical values, numerical ranges and Boolean values. Complex queries can be easily constructed from basic concepts by linking nodes together and forming query trees.

Arcs connect nodes and represent hierarchical orderings between them. Arcs emanating from a node are referred to as that node's 'outgoing arcs', and by connecting a node's outgoing arc to another node, a parent-child relationship is formed. In general, only operation nodes have outgoing arcs, since only they require input parameters to be supplied, which are represented by the nodes that connect to them. Such input parameters can either be required or optional, which is determined by the operation node that they are assigned to. Since every performance concept or operation represented by an operation node requires an input to operate on, operation nodes always have at least one outgoing arc. Nodes that are connected to operation nodes through their outgoing arcs are called 'child nodes'. Value nodes (with the exception of the Range node, which is a special case) have no outgoing arcs. Arcs are annotated with labels, which represent roles that child nodes have for their parent nodes.

The top node of a Performance Tree query represents the overall result of the query. The result's type is determined by the output type of the top node's child node. Table 3.1 provides an overview of the currently available Performance Tree operators, further details of which will be discussed in Chapter 4.

3.2 The Power of Performance Trees

The power of Performance Trees can be summarised by the following four attributes:

3.2.1 Accessibility

One of the main motivations for the development of Performance Trees was the need for an accessible performance query specification mechanism. Accessibility implies intuitive ease of use that enables users to specify performance properties in a straightforward manner.

Due to the area of application of Performance Trees, a certain amount of statistical and engineering background is necessary for the understanding and use of the formalism; however, such a background is normally characteristic of the target audience of system designers and engineers. Performance Tree queries are constructed from a set of operators that represent well-known performance concepts, and the abstract resemblance of operation nodes to functions in a programming language and value nodes to their arguments contributes to making the use of Performance Tree operators more natural.

Performance Trees ensure ease of use through their graphical nature, which allows for convenient visual composition of performance queries. Their hierarchical tree structure provides a pleasant alternative to the obfuscating nature of stochastic logic formulae, and simplifies the interpretation of queries. The ability to represent Performance Tree queries in natural language is an additional aid to the visual construction and intuitive verification of performance queries. Section 5.1 introduces a natural language-based translation mechanism that is featured in the tool supporting Performance Tree-based query specification. It provides users with continuous feedback on the natural language equivalent of queries that are in the process of construction. Due to the structure of Performance Tree

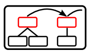
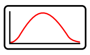
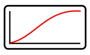
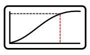
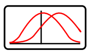
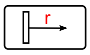
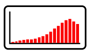
Textual	Graphical	Description
?	?	The result of a performance query.
Mult		Concurrent evaluation of multiple independent queries.
PTD		Passage time density, calculated from a given set of start and target states.
Dist		Passage time distribution obtained from a passage time density.
Perctl		Percentile of a passage time density or distribution.
Conv		Convolution of two passage time densities.
ProbInInterval	$\Pr(\bullet\xrightarrow{[t_1,t_2]}\bullet)$	Probability with which a passage takes place in a certain amount of time.
ProbInStates	$\Pr(\dashrightarrow\overset{@t}{\bullet}\dashrightarrow)$	Transient probability of a system being in a given set of states at a given point in time.
Moment	$E(X^n)$	Raw moment of a passage time density or distribution.
FR		Mean occurrence of an action (mean firing rate of a transition).
SS:P		Probability mass function yielding the steady-state probability of each possible value taken on by a StateFunc when evaluated over a given set of states.
SS:S	$\overset{\pi_x=[p_1,p_2]}{\dashrightarrow\bullet\dashrightarrow}$	Set of states that have a certain steady-state probability.
StatesAtTime	$\underset{\Pr\in[p_1,p_2]}{\overset{@t}{\dashrightarrow\bullet\dashrightarrow}}$	Set of states that the system can occupy at a given time.
InInterval	$x\in[x_1,x_2]$	Boolean operator that determines whether a numerical value is within an interval.
Macro	Macro	User-defined performance concept composed of other operators.
$\subseteq$	$\subseteq$	Boolean operator that determines whether a set is included in or corresponds to another set.
$\vee/\wedge$	$\vee/\wedge$	Boolean disjunction or conjunction of two logical expressions.
$\neg$	$\neg$	Boolean negation of a logical expression.
$\bowtie$	$<,\leq,==,\geq,>$	Arithmetic comparison of two numerical values.
$\oplus$	+,-,*,/,^	Arithmetic operation on two numerical values.
Num	Num	A real number.
Range	Range	A range of real numbers, defined by a lower and an upper bound.
Bool	Bool	A Boolean value.
States	States	A set of system states.
StateFunc	StateFunc	A real-valued function on a set of states.
Actions	Actions	A set of system actions.

Table 3.1: Description of Performance Tree operators

queries, it is also relatively simple to translate a performance query expressed in natural language into Performance Tree form. Section 5.1 provides details of a query construction mechanism supported by the tool, which allows users to define queries step-by-step in natural language, and construct the equivalent Performance Tree queries automatically.

3.2.2 Expressiveness

Perhaps the most distinctive advantage that Performance Trees have over other approaches to performance query specification is their ability to express both performance requirements and performance measures in a single query. Performance Trees are able to reason about a wide range of performance concepts from the realms of property verification and measure extraction. At present, no other formalism is able to offer similar levels of expressiveness.

As summarised in Table 3.1, Performance Trees are able to express a wide range of performance-related concepts and operations in their queries. Modellers are able to aggregate multiple independent query trees into single queries to enable parallelised evaluation that results in a significantly reduced overall computation time when compared to individual sequential query evaluations.

Performance Tree queries allow modellers to reason about response times by extracting full passage time densities and corresponding distributions from system models. Beyond the information that they already provide, passage time densities and distributions can be used to calculate percentiles, convolutions and raw moments. Operators are available that address the probabilities with which passages occur in given amounts of time, the probabilities with which systems are in sets of states at certain points in time, and the states that systems can be in at given points in time with given probabilities. Steady-state probability distributions for sets of states can be extracted from system models, and it is also possible to reason about system states that satisfy certain steady-state probability constraints. In addition, mean rates of occurrence of actions (mean firing rates of transitions in a Petri net context) can also be extracted from system models. Standard arithmetic operations and comparisons can be performed on operators that represent numerical values. Boolean operations, such as conjunctions, disjunctions and negations, can be applied to nodes that represent Boolean values, while membership operators check whether numerical values are contained within intervals, and whether sets are included in or correspond to other sets. Together, these operators ensure a high degree of sophistication in terms of reasoning ability provided to modellers.

Every performance measure query whose result is a numerical value can be transformed into a performance requirement query by inserting a relative comparison operator between the Result node and its sub-node. This has the effect of changing the return type of the query from a numerical to a Boolean value. To visualise this concept, consider the performance measure query of Figure 3.2, which considers the expected amount of time taken for some passage to occur. Figure 3.3 shows how this is converted into a performance requirement query by using the *InInterval* operator.

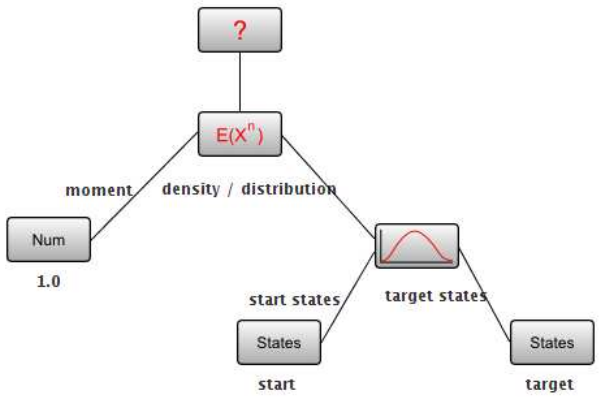


Figure 3.2: A performance measure query

Performance Trees are able to replicate most of the expressiveness of the stochastic logics CSL and eCSL through their ability to reason about model states. Paths in CSL can be represented as passages in Performance Trees by specifying start, target and excluded states. Steady-state measures can be expressed with standard Performance Tree operators. Thus, Performance Trees are able to cater for the needs of most stochastic logic-oriented users, and offer in addition significantly broader expressiveness through the availability of a wide range of miscellaneous operators.

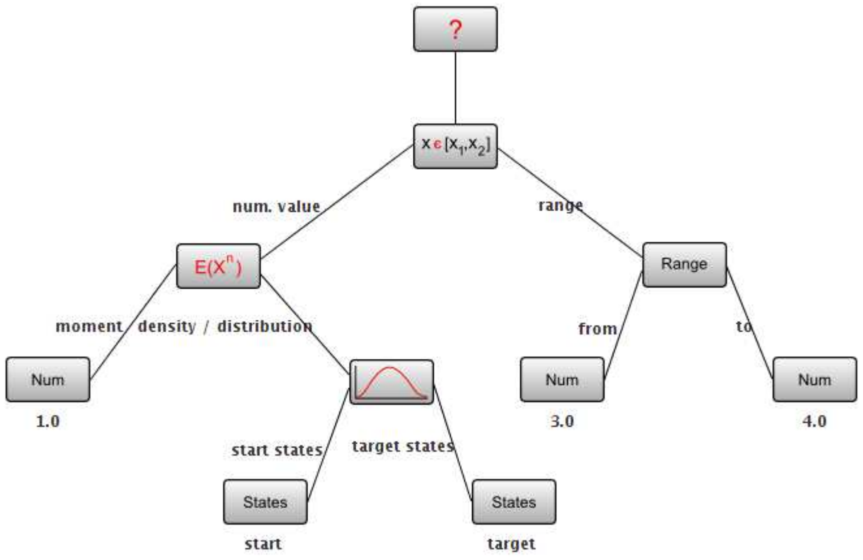


Figure 3.3: The performance measure query of Figure 3.2 converted into a performance requirement query

3.2.3 Extensibility

The Performance Tree formalism can be extended in two ways: either through the use of a macro mechanism or by the definition of new operators.

A Performance Tree macro is a shorthand representation of a user-defined performance concept. Since certain performance properties that system modellers may wish to express are not supported by the standard set of operators, Performance Trees offer the means to define such properties using a combination of standard operators. The formalism includes a special Macro operator that represents user-defined macros. In this way, performance properties that are defined by complex hierarchical tree structures can be represented by a single node in a Performance Tree query. Since macros represent performance concepts, they can be parameterised according to the needs of different analysis scenarios. The *PIPE2* tool stores macro definitions alongside model descriptions, which allows Macro nodes to be reused multiple times within the same query, or even among multiple queries on the same model. Figure 3.4 shows an example macro definition, which represents the concept of “Coefficient of Variation”.

The Coefficient of Variation is defined as the ratio of the standard deviation to the mean.

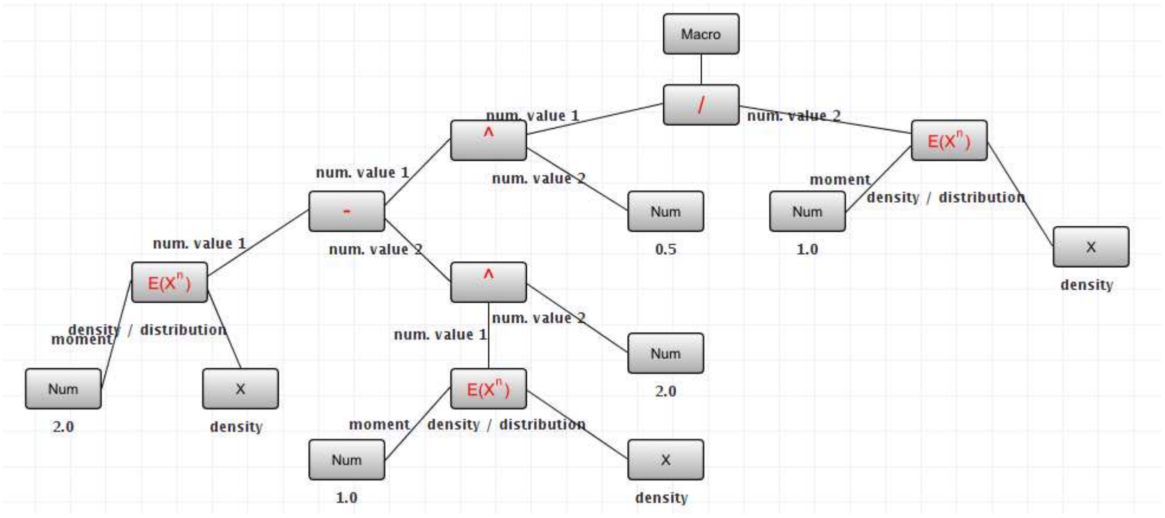


Figure 3.4: An example of Performance Tree macro expansion, used for the calculation of the Coefficient of Variation

The X node, labelled “density”, is the argument that needs to be supplied to the macro. This needs to be a passage time density, from which the standard deviation and the mean can be calculated. This argument is substituted in place of the argument nodes in the macro during evaluation. Figure 3.5 illustrates the usage of the macro in a performance query.

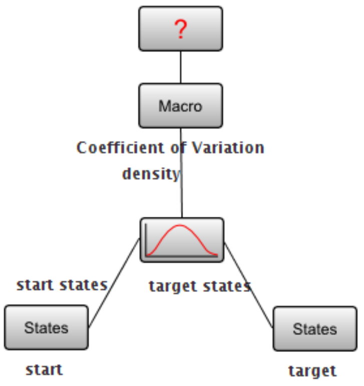


Figure 3.5: Usage of the Coefficient of Variation macro in a performance query

Another way of extending the expressiveness of Performance Trees is to incorporate operators representing new concepts into the set of standard operators. The formalism can be extended in this way without any restrictions; however, for analysis of performance queries that use such operators to be successful, tool-based evaluation support needs to be integrated into the supporting Performance Tree analysis environment (see Section 5.2).

3.2.4 Versatility

Abstract State and Action Specification

Performance Tree queries are not restricted to a single underlying modelling formalism, unlike a number of other query specification languages, such as stochastic logics for instance. Modelling formalisms reason about system states and actions differently. To ensure versatility, Performance Trees feature an abstract state specification mechanism that supports the reasoning about system states in performance queries through *state labels*. A state label is a user-defined string that identifies sets of system states through a set of associated constraints on the underlying system that are appropriate to the modelling formalism used.

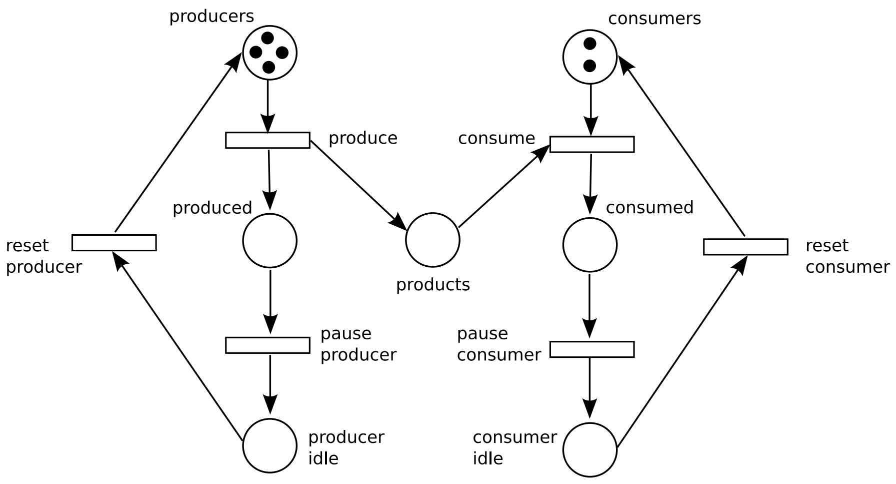


Figure 3.6: Producer-Consumer System

To visualise this concept, consider the States nodes in Figure 3.5 that have state labels *'start'* and *'target'*. For the GSPN model of a Producer-Consumer System in Figure 3.6, the modeller could define the state labels for the start and target states with the following constraints:

$$\begin{aligned} \textit{`start'} \quad &:= \quad (\#(\text{producers}) == 4) \wedge (\#(\text{consumers}) == 2) \\ \textit{`target'} \quad &:= \quad (\#(\text{products}) == 3) \end{aligned}$$

For GSPNs, a set of states can be specified using conjunctions and disjunctions of constructs of the form $(\#(\text{placename}) \bowtie x)$, where $\#(\text{placename})$ represents the number of tokens on place 'placename' and $\bowtie \in \{<, \leq, ==, \geq, >\}$. Hence, in the above state label definitions, the set of system states in which there are four tokens on place 'producers' and two tokens on place 'consumers' is identified by the state label *'start'*, while the set of states in which there are three tokens on place 'products' is identified by the state label *'target'*.

System actions are identified by *action labels*. The difference between state and action labels is that action labels correspond to the actual names of actions (or transitions in the context of SPNs) of the underlying system models. Hence, they do not use constraints for the unique identification of actions. Figure 3.7 shows how the mean rate of occurrence of action 'produce' is represented. The Actions node identifies the transition in the model through the action label *'produce'*.

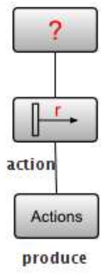


Figure 3.7: Action label example

This kind of state and action selection can be adapted to different modelling formalisms by changing constraints appropriately. This is presented in the next section in the context of SPAs.

Application to Stochastic Process Algebras

As an example of how Performance Trees can be used with modelling formalisms other than GSPNs, we present their application on a system model defined in the stochastic process algebra PEPA (see Section 2.1.3). Both GSPNs and PEPA have the ability to accurately reflect the behaviour of real life systems, with the slight difference that GSPNs are state-based, while PEPA is action-oriented. GSPNs are very good at representing the evolution of systems, while PEPA is useful when systems need to be constructed from a set of sub-components that interact in some way. A GSPN is a visual representation, while PEPA is not.

A number of Performance Tree operators reference the underlying system model directly through their sub-nodes, which can represent sets of states or individual actions. In order for Performance Tree queries to be applicable to PEPA models, they need to be able to reference states and actions unambiguously. States of PEPA models can be identified through state vectors whose elements are the counts of currently enabled components. Since PEPA is action-oriented, referencing actions in Performance Tree queries is straightforward. They can be addressed by referring to model definitions directly.

To provide an example, consider the Producer-Consumer System model of Figure 3.6. Let us translate this GSPN into PEPA by making PEPA components correspond to places of the GSPN:

$$\begin{aligned}
\mathit{producers} &\stackrel{\text{def}}{=} (\mathit{produce}, r_1).\mathit{produced} \\
\mathit{produced} &\stackrel{\text{def}}{=} (\mathit{pause\ producer}, r_2).\mathit{producer\ idle} \\
\mathit{producer\ idle} &\stackrel{\text{def}}{=} (\mathit{reset\ producer}, r_3).\mathit{producers} \\
\mathit{products}_0 &\stackrel{\text{def}}{=} (\mathit{produce}, r_4).\mathit{products}_1 \\
\mathit{products}_n &\stackrel{\text{def}}{=} (\mathit{produce}, r_4).\mathit{products}_{n+1} \\
\mathit{products}_n &\stackrel{\text{def}}{=} (\mathit{consume}, r_5).\mathit{products}_{n-1} \\
\mathit{consumers} &\stackrel{\text{def}}{=} (\mathit{consume}, r_6).\mathit{consumed} \\
\mathit{consumed} &\stackrel{\text{def}}{=} (\mathit{pause\ consumer}, r_7).\mathit{consumer\ idle} \\
\mathit{consumer\ idle} &\stackrel{\text{def}}{=} (\mathit{reset\ consumer}, r_8).\mathit{consumers}
\end{aligned}$$

The system equation is given by:

$$(\mathit{producers}[4] \mid\mid \mathit{consumers}[2]) \underset{\{produce,consume\}}{\bowtie} \mathit{products}_0$$

We can now identify a set of states uniquely by defining a state vector that contains all enabled components of the model, as shown in the example below:

$($*producers, producers, produced, producer idle, consumers, consumer idle, products*$_1)$

This state vector identifies a set of states in which two *producers*, one *produced*, one *producer idle*, one *consumers*, one *consumer idle* and one *products*$_1$ components are enabled. Since potentially there may be a large number of enabled components in a PEPA model, it is possible to use a more convenient alternative notation for specifying sets of states for PEPA models. In line with the approach presented in [Hillston05], we aggregate the model and represent its states in numerical vector form. Such a vector is of length equal to that of the state vector, i.e. the total number of components of the system, and has integer elements, each of which indicates the number of a particular enabled component. Given that the state vector $($*producers, produced, producer idle,* $\dots)$ contains all components that can possibly be enabled, the corresponding numerical vector for the example above would be $(2, 1, 1, \dots)$.

Support for Customer Tracking

The dynamic behaviour of GSPNs is characterised by the creation and destruction of tokens representing customers and resources in systems. Tokens are indistinguishable from one another, which implies that it is often not possible to express performance properties that reason about individual customers. However, it is important to be able to express such properties, since questions of the type *"Is the probability of a customer being served within 1 minute greater than 95%?"* often arise in QoS requirements.

For GSPN-based system analyses, Performance Trees offer the ability to reason about tagged customers [Dingle08a] by using an approach derived from the QN tagged tokens technique of [Mitrani98]. To use tagged tokens in GSPNs, certain arcs of the GSPN need to be tagged to indicate the permissible flow of tagged tokens. In a GSPN, there can only ever be a single tagged token, which represents the individual customer that is being observed. A GSPN with a tagged token contains two types of arcs: regular and tagged arcs. Tagged tokens may only flow along tagged arcs, whereas normal tokens can be transported along both regular and tagged arcs. Additional structural restrictions require that tagged arcs only have weights of 1, and that any transition that has a tagged input arc should also have a tagged output arc in order to preserve the tagged token. To incorporate into performance queries properties that relate to the position of the tagged token, the *(tag@(placename))* construct is used in the definition of state labels to identify all system states in which the tagged token is located at place 'placename'. Analysis tools forming part of the evaluation environment (see Section 5.2) for Performance Trees incorporate tagged token-based analysis capabilities.

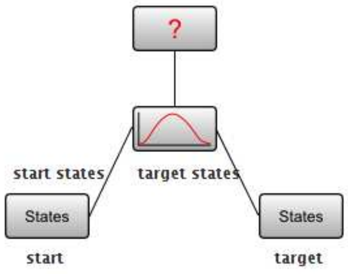


Figure 3.8: Tagged customer query

For our running example, consider a simple passage time density query of the form of Figure 3.8. Here, we wish to express the query *"In the Producer-Consumer System, what is the density of time taken to reach a system state where a tagged producer is idle and there are no consumers available, provided that the system has started in a state where the tagged producer was available, along with three other producers, as well as two consumers?"* This query necessitates the ability to reason about an individual customer. Hence, we define appropriate start and target states for the passage as follows:

$$\begin{aligned} \text{`}\textit{start}\text{'} &:= (\text{tag@(producers)}) \wedge (\#(\text{producers}) = 4) \wedge (\#(\text{consumers}) = 2) \\ \text{`}\textit{target}\text{'} &:= (\text{tag@(producer idle)}) \wedge (\#(\text{consumers}) = 0) \end{aligned}$$

Section 4.3 provides a detailed explanation of the abstract state specification mechanism during the discussion of the States and StateFunc operators. See [Dingle08a] for a full description of the semantics for and analysis of tagged tokens in GSPN models.

Customer tracking in PEPA is realised by adding location-awareness to stochastic probe specifications [Argent-Katwala07a], which provides the ability to identify individual model components within PEPA models for selective instrumentation. Details of this technique are given in [Argent-Katwala07b].

3.3 Performance Trees in Action

To demonstrate the applicability of Performance Trees in practice, let us consider an online transaction system that serves as a company's electronic retailing platform for selling products through their corporate web site. A GSPN model of this system is shown in Figure 3.9 and operates as follows.

Customers arrive at the web site with a certain rate, and proceed to browse the online product catalogue. At any point, they can decide to leave the web site and navigate to an unrelated page on the Web. While browsing the catalogue, customers can select items or jump straight to the checkout page, in case they are returning customers who have already selected an item previously. Once an item has been selected, customers can either proceed to the checkout or continue to browse the catalogue.

At the checkout, they are either required to register if they are visiting the web site for the first time, or to log in if they happen to be returning customers. Alternatively, they can decide to return to the product catalogue to search for more items before proceeding to the placement of an order. Once customers have registered or logged in, they are asked to provide the address to which the order is to be delivered, as well as the billing details. Once an order has been confirmed, customers are taken back to the product catalogue.

Below, we demonstrate the expressive power of Performance Trees by posing a number of performance queries in natural language and expressing them as Performance Trees.

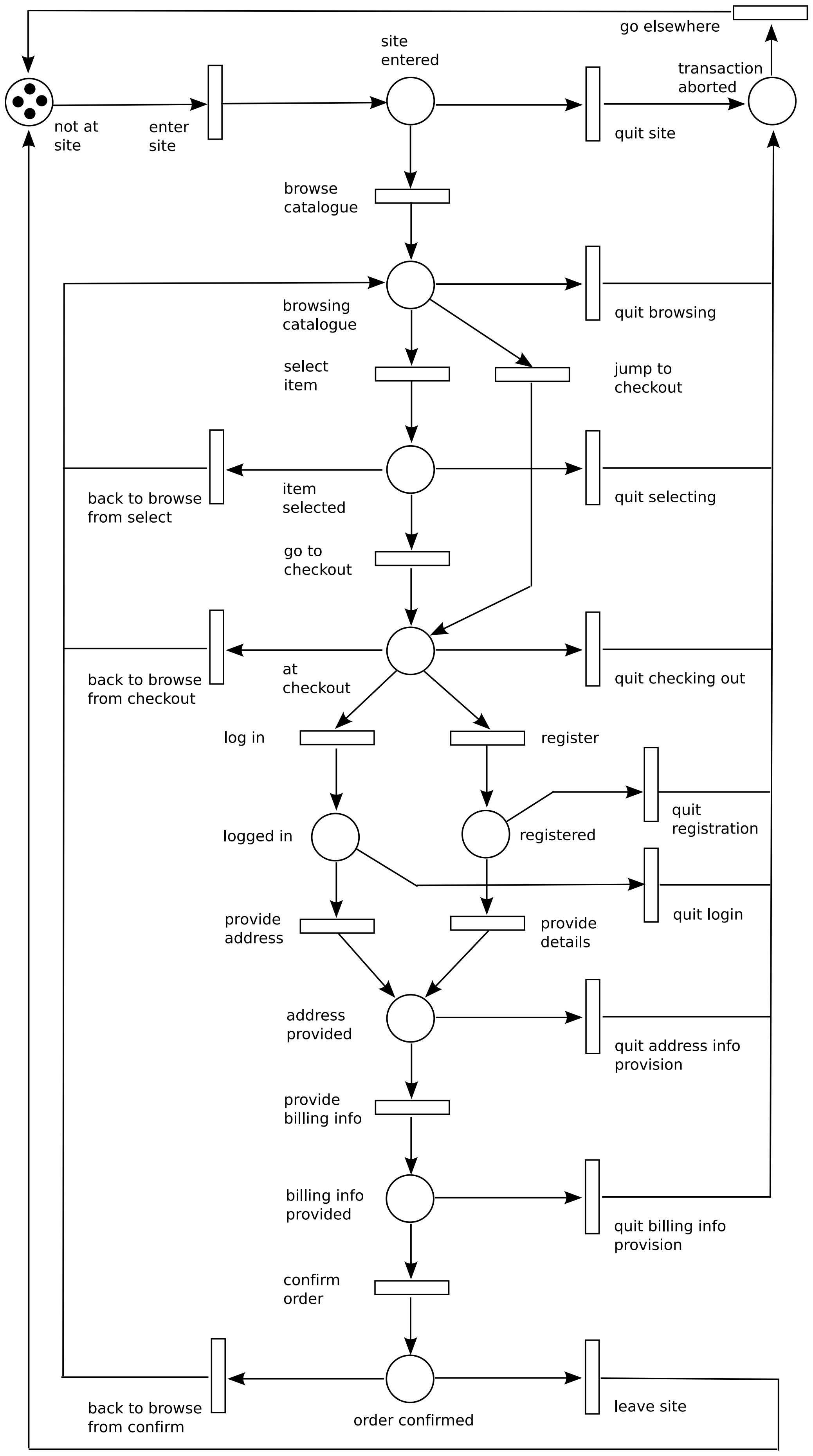


Figure 3.9: An Online Transaction System

NL Query: *"From the moment that a customer has entered the web site for the first time, what is the distribution of time required for them to select an item from the product catalogue?"*

PT Query: Shown in Figure 3.10, with relevant state labels defined as:

'start'	:=	(tag@(site entered))
'target'	:=	(tag@(item selected))
'excluded'	:=	(tag@(transaction aborted))

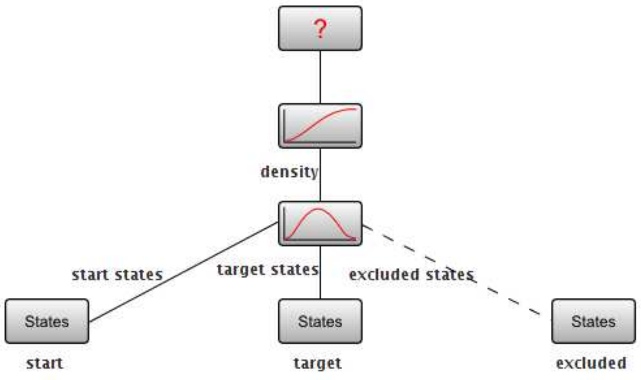


Figure 3.10: A query addressing a passage time distribution

NL Query: *"What is the probability of some product having been ordered within 20 minutes after 4 customers entering the site at the same time, and none having left in the meantime?"*

PT Query: Shown in Figure 3.11, with relevant state labels defined as:

'start'	:=	$(\#(\text{site entered}) == 4) \wedge (\#(\text{order confirmed}) == 0)$
'target'	:=	$(\#(\text{order confirmed}) \geq 1)$
'excluded'	:=	$(\#(\text{transaction aborted}) \geq 1)$

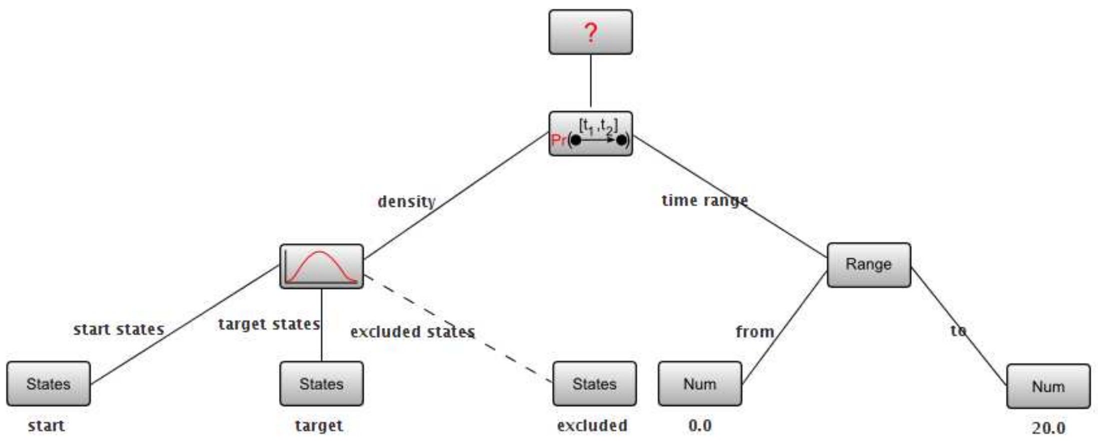


Figure 3.11: A query addressing the probability of a passage occurring within a time interval

NL Query:	*"Is the 97th percentile of the convolution of two passage time densities less than 4.71 minutes, where the first passage is considered from the point when a customer has entered the web site to when they are at the checkout, and the second passage is considered from the moment that a user has provided their address to when they have confirmed the order, assuming that all throughout the user has not aborted the transaction?"*
PT Query:	Shown in Figure 3.12, with relevant state labels defined as:

'start1'	:=	(tag@(site entered))
'target1'	:=	(tag@(at checkout))
'start2'	:=	(tag@(address provided))
'target2'	:=	(tag@(order confirmed))
'excluded'	:=	(tag@(transaction aborted))

NL Query:	*"Is the probability of at least 2 customers being at the checkout at time instant 14 minutes greater than 74%, provided that 4 customers have entered the site at time 0?"*
PT Query:	Shown in Figure 3.13, with relevant state labels defined as:

'start'	:=	(#(site entered) == 4)
'target'	:=	(#(at checkout) $\geq$ 2)

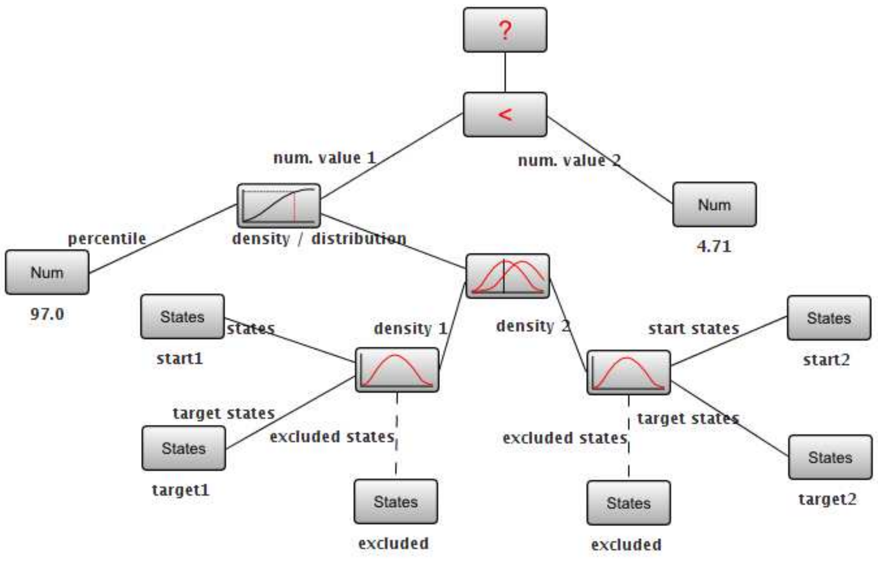


Figure 3.12: A query addressing the percentile of a convolution of two passage time densities

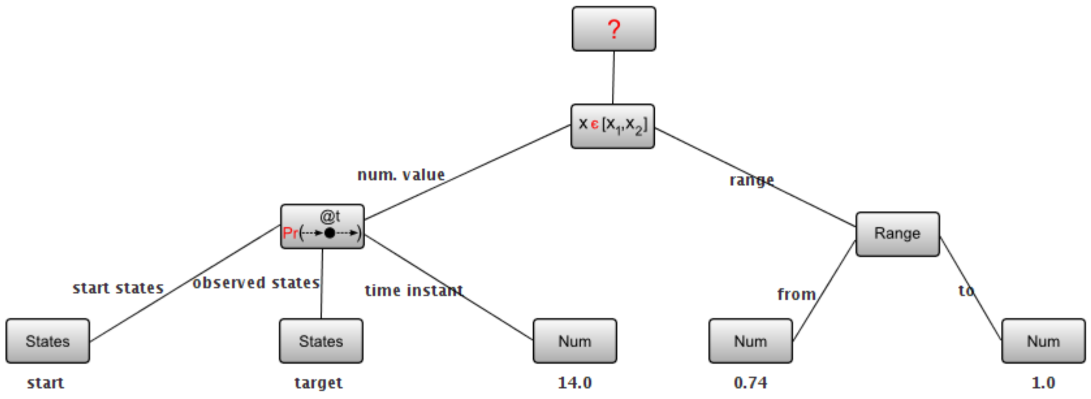


Figure 3.13: A query addressing the transient probability of a system being in a given state at a given time

NL Query:	*"Are the states that the system can be in at time instant 36 minutes with at least 80% probability contained within the set of states in which there are no customers browsing other web sites?"*
PT Query:	Shown in Figure 3.14, with relevant state labels defined as:

'all customers on web site' := (#(not at site) == 0)

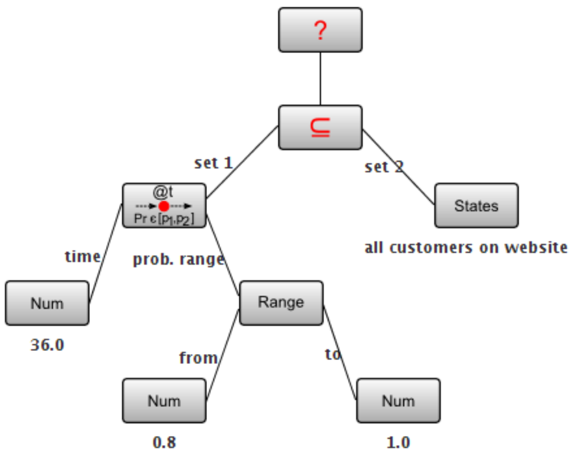


Figure 3.14: A query addressing the states that the system can occupy at a given time with some probability

NL Query: *"What is the average rate of customers entering the site; what is the steady-state probability distribution of the number of customers at the checkout; and which states of the system have a steady-state probability greater than 0.2, given that the system has started in a state in which 4 customers were browsing other web sites at time 0?"*

PT Query: Shown in Figure 3.15

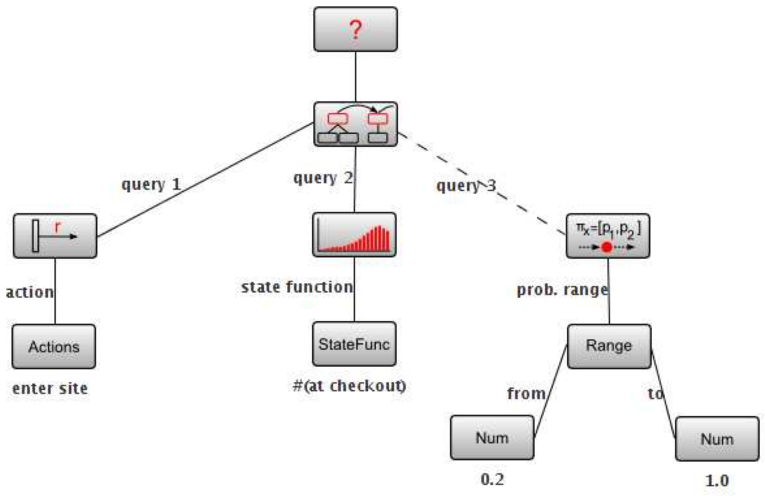


Figure 3.15: A query aggregating multiple independent queries that address the average occurrence of an action, the steady-state probability of the system, and states that conform to a certain steady state probability requirement

This query makes use of the Mult operator to combine multiple independent performance queries into one. The notation '#(at checkout)' annotating the StateFunc node represents the number of tokens on place 'at checkout' in the context of the GSPN model of the Online Transaction System. Details of its use are given along with the explanation of the semantics of the StateFunc node in Section 4.3.

To demonstrate that Performance Trees are also able to specify basic arithmetic queries, consider the following:

NL Query:	*"What is 5 minus 3 multiplied by 6 and raised to the power of 2?"*
PT Query:	Shown in Figure 3.16

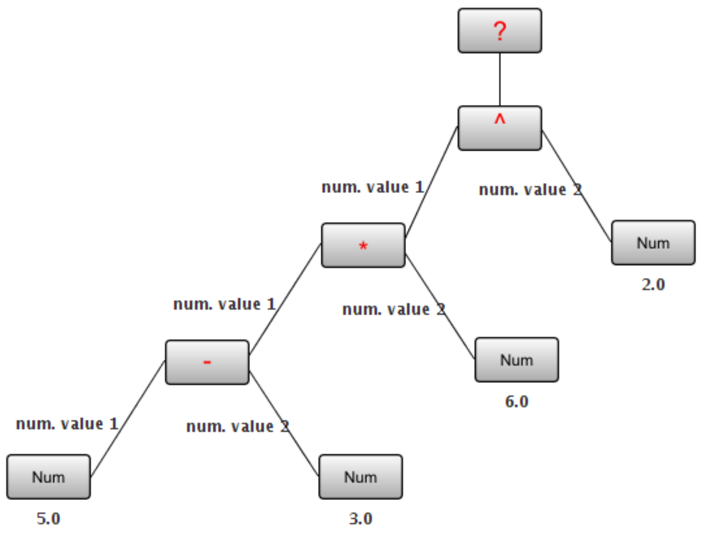


Figure 3.16: A query addressing basic arithmetic operations

Chapter 4

Formal Characterisation of Performance Trees

In this chapter, we present a formal characterisation of the Performance Tree formalism. We describe its syntax, which defines the valid use of Performance Tree operators, and provide a textual representation framework that serves as an alternative to graphical query specification. We discuss the typing of Performance Tree operators, used to verify the type safety of node assignments within queries, and detail quantitative semantics, which define the mathematical interpretation of operators. This chapter does not aim at suggesting or describing specific solution strategies or algorithms for implementers, but rather serves the purpose of rigorously clarifying the semantics of Performance Tree operators in the context of (semi-)Markovian modelling formalisms. It builds on material presented in [Suto06b] and [Suto07].

4.1 Syntax

The syntax describes the nature of Performance Tree value nodes and defines possible sub-node types for Performance Tree operators, thereby establishing rules according to which performance queries are constructed. In the textual notation, individual sub-nodes of operators are separated by commas, multiple instances of sub-nodes are represented by the '^{n}' superscript notation, and choice is indicated by ' | '.

4.1.1 Value Node Syntax

The **Num** node represents a real value. Depending on which operation node is its parent within a query, the value it represents is interpreted differently. It can represent a real number, a percentile, the rank of a moment, a probability or a time value.

The **Range** node represents a range of real numbers. It is an exception to the convention of value nodes not having sub-nodes, as it requires two sub-nodes to define the range that it represents. Nevertheless, it is considered a value node, due to the fact that it is simply a basic type consisting of a pair of real numbers.

$$\text{Range} \quad := \quad (\text{Num})^2$$

The **Bool** node represents a Boolean value, i.e. *true* or *false*.

The **States** node represents a set of states. State labels are used for identifying sets of states of the model. Section 4.3 also gives details on the state labelling mechanism.

The **StateFunc** node represents a user-defined real-valued function on a set of states. As a sub-node of the SS:P operator, it is used to define criteria based on which a steady-state probability distribution is to be calculated.

The **Actions** node represents a set of actions. As mentioned earlier, action labels are used for the identification of individual actions within the model. Details of the action labelling mechanism are given in Section 4.3.

4.1.2 Operation Node Syntax

The **?** operator represents the result of a performance query, and is the topmost node of every query tree. If its sub-node is an operation or value node, it represents whatever the sub-node evaluates to. However, in case its sub-node is a Mult node, it represents the set of values to which the Mult node's sub-nodes evaluate.

? $:=$ Mult | PTD | Dist | Perctl | Conv | ProbInInterval | ProbInStates | Moment | FR | SS:P | SS:S | StatesAtTime | InInterval | Macro | $\subseteq$ | $\vee/\wedge$ | $\neg$ | $\bowtie$ | $\oplus$ | Num | Range | Actions | States | Bool

The **Mult** operator allows multiple independent queries to be defined simultaneously and combined into a single performance query. It represents the set of results that these queries evaluate to individually. The operator requires at least two, but can have arbitrarily many, sub-nodes.

Mult $:=$ (PTD | Dist | Perctl | Conv | ProbInInterval | ProbInStates | Moment | FR | SS:P | SS:S | StatesAtTime | InInterval | Macro | $\subseteq$ | $\vee/\wedge$ | $\neg$ | $\bowtie$ | $\oplus$ | Num | Range | Actions | States | Bool)$^{2..*}$

The **PTD** operator represents the density of time that elapses during a passage between two sets of states. To specify the passage, the set of start and the set of target states need to be provided as sub-nodes. Optionally, a further sub-node may be provided to represent the set of states that are excluded from the passage.

PTD $:=$ (States | SS:S | StatesAtTime)$^{2..3}$

The **Dist** operator represents the cumulative distribution function corresponding to an underlying density function. The distribution is obtained from the passage time density or convolution of two passage time densities (which in terms of its type is also a density) that is represented by the sub-node.

Dist $:=$ PTD | Conv

The **Perctl** operator represents a percentile of a passage time density or distribution. The operator requires two sub-nodes: one sub-node specifying which percentile is to be calculated, and the other sub-node representing a density, distribution or convolution of two passage time densities.

Perctl	:=	Num, (PTD | Conv | Dist)

The **Conv** operator represents the convolution of two independent probability density functions into a single density function. The operator has two sub-nodes, which can either be passage time densities or convolutions themselves.

Conv	:=	(PTD, PTD) | (Conv, PTD) | (Conv, Conv)

The **ProbInInterval** operator represents the probability with which a passage is completed in a certain amount of time. The operator has two sub-nodes: a passage time density or a convolution, which defines the passage, and a time range, which defines the time interval during which the passage is to be completed.

ProbInInterval	:=	(PTD | Conv), Range

The **ProbInStates** operator represents the probability of a system being in a set of states at a particular instant in time, given that it has started in some initial state at time 0. The operator has two sub-nodes, the first of which represents the set of states that the system occupies at the observation instant, and the second of which represents the time instant of interest.

ProbInStates	:=	(States | SS:S | StatesAtTime), Num

The **Moment** operator represents a raw moment of a passage time density. A moment generating function can provide us with multiple metrics, since we can derive any number of central moments, which can provide valuable insight into the nature of the passage. The operator has two sub-nodes. The first sub-node specifies which moment is to be calculated (e.g. the 3^{rd} moment), while the second sub-node specifies the passage time density, distribution or convolution that the moment is calculated from.

Moment $:=$ Num, (PTD | Conv | Dist | SS:P)

The **FR** operator represents the average rate of occurrence of a set of actions. The operator's sub-node specifies the actions of interest.

FR $:=$ Actions

The **SS:P** operator represents a probability mass function yielding the steady-state probability of each possible value taken on by a StateFunc when evaluated over a given set of states. The operator has one required and one optional sub-node. The required sub-node represents a state function, which imposes constraints on the calculation of the steady-state distribution. The optional argument represents the set of states, over which the distribution is to be calculated. If this sub-node is not provided, the calculation defaults to the entire state space of the model.

SS:P $:=$ StateFunc, (States | SS:S | StatesAtTime)

The **SS:S** operator represents the set of states whose steady-state probability lies within a certain range. The operator has a single sub-node, which defines the acceptable range for the steady-state probability.

SS:S $:=$ Range

The **StatesAtTime** operator represents the set of states that the system can occupy at a given time instant with probability lying within a certain range, given that it has started in an initial state at time 0. The operator has two sub-nodes: one representing the time instant of interest and one representing the probability range of relevance.

StatesAtTime $:=$ Num, Range

The **InInterval** operator represents the Boolean value that determines whether a numerical value is within a certain interval. It has two sub-nodes: one representing the numerical value that is to be tested, and one representing the numerical range that is to be tested against.

InInterval := (Perctl | ProbInInterval | ProbInStates | Moment | FR | $\oplus$), Range

The **Macro** operator represents a user-defined performance concept that is defined by the composition of basic Performance Tree operators. A Macro node has as many arguments as the macro that it represents. Whenever Macro nodes represent concepts that are appropriate as arguments to other operators, they can be used as sub-nodes.

Macro := (PTD | Dist | Perctl | Conv | ProbInInterval | ProbInStates | Moment | FR | SS:P | SS:S | StatesAtTime | InInterval | Macro | $\subseteq$ | $\vee/\wedge$ | $\neg$ | $\bowtie$ | $\oplus$ | Num | Range | Actions | States | Bool)$^{0..*}$

The $\subseteq$ operator represents a Boolean value that expresses whether a particular set of states is included in or corresponds to another set of states. Therefore, it has two sub-nodes, both of which represent sets of states.

$\subseteq$:= (States | SS:S | StatesAtTime)^2

The $\vee/\wedge$ operator represents a Boolean disjunction or conjunction operation. It has two sub-nodes, both of which represent Boolean values.

$\vee/\wedge$:= (InInterval | $\subseteq$ | $\vee/\wedge$ | $\neg$ | $\bowtie$ | Bool)^2

The $\neg$ operator represents a Boolean negation. It has a single sub-node, which represents the Boolean value that is to be negated.

$\neg$:= InInterval | $\subseteq$ | $\vee/\wedge$ | $\neg$ | $\bowtie$ | Bool

The $\bowtie$ operator represents an arithmetic comparison. $\bowtie \in \{<, \leq, ==, \geq, >\}$. The operator has two sub-nodes, both of which represent the numerical values that are to be compared.

$\bowtie$:= (Perctl | ProbInInterval | ProbInStates | Moment | FR | $\oplus$ | Num)^2

The $\oplus$ operator represents an arithmetic operation. $\oplus \in \{+, -, *, /, \hat{}\}$. Therefore, both sub-nodes of the operator represent numerical values.

$\oplus$:= (Perctl | ProbInInterval | ProbInStates | Moment | FR | $\oplus$ | Num)^2

4.1.3 Textual Representation

Performance Tree queries are mainly specified graphically, but they can also be represented in a textual manner. A textual representation can be practical when writing out queries, in which case graphical query descriptions may be impractical. Performance Tree nodes are represented textually as tuples of the form:

(role, node(...))

where *node* is the textual form of the Performance Tree node being represented, as specified in Table 3.1. Every node has an associated role, which is represented by the string *role*. Roles are used by operator nodes to classify sub-nodes. If, for example, a PTD node has multiple sub-nodes of type States, a distinction needs to be made regarding the sub-node representing the set of start states, and the node representing the set of target states. In the graphical representation, the label of an arc connecting two nodes indicates the role that the sub-node has from its parent node's point of view. The only operator node that does not have a role is the ? node.

In the case of operation nodes, hierarchies apparent in the graphical representation are translated into textual form by using the above notation and specifying sub-nodes within enclosing brackets of a node:

(role, opnode((role, subnode(. . .)),. . .))

For value nodes, the textual form is as follows:

(role, valnode(value))

where *value* represents the numerical or Boolean value, state or action label, or state function that is represented by the value node. The Range node, however, is an exception. Even though it is considered a value node, it requires sub-nodes to define the numerical range. Therefore, it is represented in the same way as an operation node.

To visualise this concept, consider the Performance Tree query of Figure 3.15. It can be expressed in textual form as:

```
?(query, Mult(
      (query1, FR((action, Actions(enter site)))),
      (query2, SS:P((state function, StateFunc(#(at checkout))))),
      (query3, SS:S((prob. range, Range((from, Num(0.87)),
                    (to, Num(1.0))))))))
```

4.2 Typing

Every Performance Tree node within a query has a type, and the structure of Performance Tree queries is dependent on type compatibility between nodes. It is therefore important to have formal typing defining the rules according to which queries can be constructed. The following domains are used for the description of Performance Tree operators:

$\mathcal{S}$	:	the state space (set of all states) of a model
$\mathcal{A}$	:	the set of actions of a model
AP	:	the set of atomic propositions
$\mathcal{B}$	$=$	$\{$*true, false*$\}$

We consider the following basic types for Performance Trees:

num	:	$x \in \mathbb{R}$, i.e. a real value
range	:	$[x, y] \in \mathbb{R} \times \mathbb{R}$, i.e. a range of real values
bool	:	$b \in \mathcal{B}$, i.e. a Boolean value
actions	:	$\{a : \mathcal{A}\}$, i.e. a set of actions
states	:	$\{s : \mathcal{S}\}$, i.e. a set of states
statefunc	:	$(AP \rightarrow \mathbb{R})$, i.e. a real-valued function on a set of states identified by a state label
probfunc	:	$(\mathbb{R} \rightarrow \mathbb{R})$, i.e. a probability distribution or density function
mult	:	$\{$(*num* $\mid$ *range* $\mid$ *bool* $\mid$ *actions* $\mid$ *states* $\mid$ *probfunc*)$^{2..*}\}$

With these basic types in mind, we introduce the notation:

$$\textit{node} \quad \vdash \quad (\textit{subnodetype}) : \textit{nodetype}$$

which expresses that a valid sub-node of the Performance Tree operator *node* has the type *subnodetype*, and that *node* represents an operation whose result has the type *nodetype*. Using this notation, we define the typing for Performance Trees as follows:

?	$\vdash$	(*num* $\mid$ *range* $\mid$ *bool* $\mid$ *actions* $\mid$ *states* $\mid$ *probfunc* $\mid$ *mult*) : (*num* $\mid$ *range* $\mid$ *bool* $\mid$ *actions* $\mid$ *states* $\mid$ *probfunc* $\mid$ *mult*)
Mult	$\vdash$	(*num* $\mid$ *range* $\mid$ *bool* $\mid$ *actions* $\mid$ *states* $\mid$ *probfunc*)$^{2..*}$: *mult*
PTD	$\vdash$	(*states*)$^{2..3}$: *probfunc*
Dist	$\vdash$	(*probfunc*) : *probfunc*
Perctl	$\vdash$	(*num* $\times$ *probfunc*) : *num*
Conv	$\vdash$	(*probfunc*)^{2} : *probfunc*
ProbInInterval	$\vdash$	(*probfunc* $\times$ *range*) : *num*
ProbInStates	$\vdash$	(*states* $\times$ *num*) : *num*
Moment	$\vdash$	(*num* $\times$ *probfunc*) : *num*
FR	$\vdash$	(*actions*) : *num*

SS:P	$\vdash$	*(statefunc $\times$ states)* : *probfunc*
SS:S	$\vdash$	*(range)* : *states*
StatesAtTime	$\vdash$	*(num $\times$ range)*: *states*
InInterval	$\vdash$	*(num $\times$ range)* : *bool*
Macro	$\vdash$	*(num $\mid$ range $\mid$ bool $\mid$ actions $\mid$ states $\mid$ probfunc)*$^{0..*}$: *(num $\mid$ range $\mid$ bool $\mid$ actions $\mid$ states $\mid$ probfunc)*
$\subseteq$	$\vdash$	*(states)*^{2} : *bool*
$\vee/\wedge$	$\vdash$	*(bool)*^{2} : *bool*
$\neg$	$\vdash$	*(bool)* : *bool*
$\bowtie$	$\vdash$	*(num)*^{2} : *bool*
$\oplus$	$\vdash$	*(num)*^{2} : *num*
Num	$\vdash$	*()* : *num*
Range	$\vdash$	*(num)*^{2} : *range*
Bool	$\vdash$	*()* : *bool*
Actions	$\vdash$	*()* : *actions*
States	$\vdash$	*()* : *states*
StateFunc	$\vdash$	*()* : *statefunc*

4.3 Quantitative Semantics

This section describes the formal mathematical meaning underlying Performance Tree operators. This meaning is presented in the context of (semi-)Markov models and Laplace transforms where applicable. Efficient and/or scalable algorithmic implementations are available for most operators, and are presented in [Knottenbelt96, Dingle03, Dingle04b, Bradley03a, Bradley03b, Bradley03d, Bradley04, Au-Yeung04].

Throughout this section, we will adhere to the following notational conventions:

- Scalar values are denoted by lowercase and sets by uppercase letters.
- *Eval*(*node*) is a function that evaluates *node*, a Performance Tree node, and returns the result.

- $\mathcal{L}_S : \mathcal{S} \rightarrow 2^{AP}$ is a labelling function that assigns atomic proposition labels from AP to a state from $\mathcal{S}$, i.e. $\mathcal{L}(s)$ returns the set of labels that are associated with state s.
- $\mathcal{L}_A : \mathcal{A} \rightarrow AP$ is a labelling function that assigns a label (an atomic proposition) from AP to an action from $\mathcal{A}$, i.e. $\mathcal{L}(a)$ returns the label that is associated with action a.
- $Rng(R)$ is a function that converts the numerical range $R = [x, y]$ into the Performance Tree representation "Range((from, Num(x)), (to, Num(y)))".

4.3.1 Value Node Semantics

Num operator

The Num operator represents a real value. Its evaluation yields:

$$\textit{Eval}(\text{Num}(\textit{numVal})) = \textit{numVal}$$

where *numVal* $\in \mathbb{R}$.

Range operator

The Range operator represents a range of real values. Its evaluation yields:

$$\textit{Eval}(\text{Range}((\text{from}, \textit{subnode1}(\text{A})), (\text{to}, \textit{subnode2}(\text{B})))) = [r_1, r_2]$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1. We also have that *Eval*(*subnode1*(A)) $= r_1$ and *Eval*(*subnode2*(B)) $= r_2$.

Bool operator

The Bool operator represents a Boolean value. Its evaluation yields:

$$\mathit{Eval}(\text{Bool}(\mathit{boolVal})) \quad = \quad \mathit{boolVal}$$

where *boolVal* $\in \mathcal{B}$.

States operator

The States operator represents a set of system states.

A number of Performance Tree operators have sub-nodes that represent sets of states. Therefore, an elegant and formalism-independent way of specifying model states is needed. Our approach to state identification uses atomic propositions that can be combined with Boolean connectives. A valid composition α of atomic propositions is given by:

$$\alpha \quad \overset{\text{def}}{=} \quad \mathit{true} \ \mid a \ \mid \neg\, \alpha \ \mid \alpha \wedge \alpha$$

where $a \in AP$ is used as a state label (see Section 3.2.4). The conjunction and/or disjunction of state labels L identifies a set of states for which $\{s \in \mathcal{S} : s \models L\}$, given that the semantics of $s \models L$ are defined as:

$$\begin{array}{llll} s \models \mathit{true} & \text{for all } s & s \models \neg A & \text{iff } s \not\models A \\ s \models A & \text{iff } A \in \mathcal{L}_S(s) & s \models A \wedge B & \text{iff } s \models A \ \wedge \ s \models B \end{array}$$

Thus, the evaluation of the operator yields:

$$\mathit{Eval}(\text{States}(\mathit{stateLabel})) \quad = \quad \{s\}$$

where $s \in \mathcal{S}$ and $s \models$ *stateLabel*.

StateFunc operator

The StateFunc operator represents a user-defined function on a set of states, which returns a real value.

The syntax of state function A is defined as:

$$\mathrm{A} \;\overset{\text{def}}{=}\; r \mid f(\mathrm{A}) \mid \mathrm{A} \;\; op \;\; \mathrm{A}$$

where $r \in \mathbb{R}$, $f : (\mathbb{R} \to \mathbb{R})$ is a user-defined real-valued function and *op* $\in \{+, -, *, /, <, \leq, ==, \geq, >\}$. The semantics of a state function A are defined in terms of the function *sfEval*(A, s), which calculates the value of state function A for a particular state s:

$$\begin{aligned}
\textit{sfEval}(r, s) &= r \quad \text{for all s} \\
\textit{sfEval}(f(\mathrm{A}), s) &= f(\textit{sfEval}(\mathrm{A}, s)) \\
\textit{sfEval}(\mathrm{A}_1 \;\; op \;\; \mathrm{A}_2, s) &= \textit{sfEval}(\mathrm{A}_1, s) \;\; op \;\; \textit{sfEval}(\mathrm{A}_2, s)
\end{aligned}$$

The expression $\#(a)$, where a represents an expression derivable from a state vector, as appropriate to the modelling formalism used, is a special instance of $f(A)$ that is interpreted slightly differently, depending on the underlying modelling formalism. Its context-dependent evaluation is defined as:

$$\begin{aligned}
\textit{sfEval}(\#(a), s) &= \text{the number of tokens on place } a \text{ in state } s \\
& \text{(if model is a GSPN)} \\
\textit{sfEval}(\#(a), s) &= \text{the number of } a \text{ components in state } s \\
& \text{(if model is PEPA)} \\
\textit{sfEval}(\#(a), s) &= \text{the number of customers in the queue at server } a \text{ in state } s \\
& \text{(if model is a QN)}
\end{aligned}$$

Thus, the evaluation of the operator yields:

$$\textit{Eval}(\text{StateFunc}(\textit{statefunction})) \quad = \quad \textit{sfEval}(\textit{statefunction}, s)$$

- for all $s \in \mathcal{S}$ in the context of SS:P((*state function*, StateFunc(*statefunction*)))
- for all $s \in$ *Eval*(States(*states*)) in the context of SS:P((*state function*, StateFunc(*statefunction*)), (*states*, States(*states*)))

Actions operator

The Actions operator represents a set of system actions. Actions are identified by action labels, and the evaluation of the operator yields:

$$\mathit{Eval}(\text{Actions}(\mathit{actionLabel})) \quad = \quad \{a\}$$

where $a \in \mathcal{A}$ and *actionLabel* $= \mathcal{L}_A(a)$.

4.3.2 Operation Node Semantics

? Operator

Unlike other Performance Tree operators (with the exception of the Mult operator), the ? operator does not represent an operation, but rather the value that its subnode evaluates to. Evaluation of the operator yields:

$$\mathit{Eval}(?((\text{query}, \mathit{subnode}(\text{A})))) \quad = \quad \mathit{Eval}(\mathit{subnode}(\text{A}))$$

where *subnode* is a valid subnode of the operator, according to the syntax of Section 4.1.

Mult Operator

Similarly to the ? operator, the Mult operator does not represent an operation, but rather a vector of the values that its sub-nodes evaluate to. Evaluation of the operator yields:

$$\mathit{Eval}(\text{Mult}((\text{query1}, \mathit{subnode1}(\text{A})), (\text{query2}, \mathit{subnode2}(\text{B})), \ldots)) = \\ (\mathit{Eval}(\mathit{subnode1}(\text{A})), \mathit{Eval}(\mathit{subnode2}(\text{B})), \mathit{Eval}(\ldots))$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1.

PTD Operator

The PTD operator represents a passage time density function. Its evaluation yields:

$$\mathit{Eval}(\text{PTD}((\text{start}, \mathit{subnode1}(\text{A})), (\text{target}, \mathit{subnode2}(\text{B})), (\text{excluded}, \mathit{subnode3}(\text{C})))) = \\ f_{IJK}(t)$$

where *subnode1* and *subnode2* are required sub-nodes of the operator, and *subnode3* is an optional subnode of the operator, according to the syntax of Section 4.1. $f_{IJK}(t)$ is the probability density function of P_{IJK}, the first passage time from a set of source states $I = \{s \in \mathcal{S} : s \models A\}$ to a set of target states $J = \{s \in \mathcal{S} : s \models B\}$, provided that the set of excluded states $K = \{s \in \mathcal{S} : s \models C\}$ is not encountered along the passage. That is, the first time the system enters a state in J, given that it has started in one of the states in I and has not been in any state in K, and at least one transition has occurred. In the context of (semi-)Markov processes, this is defined as:

$$P_{IJK} = \inf\{t > 0 : Z(t) \in J, Z(0) \in I, \forall k. 0 < k \leq t(Z(k) \notin K), N(t) > 0\}$$

where $Z(t)$ is a Markov renewal process, and $N(t)$ is the number of state transitions that have taken place by time t.

Dist operator

The Dist operator represents a passage time distribution function corresponding to a passage time density. Its evaluation yields:

$$\textit{Eval}(\text{Dist}((\text{density}, \textit{subnode}(\text{A})))) \quad = \quad F(t) = \int_0^t f_{\text{A}}(\tau)\, d\tau$$

where *subnode* is a valid subnode of the operator, according to the syntax of Section 4.1, and $f_{\text{A}}(t)$ is the pdf of P_{A}, the random variable corresponding to the first passage time of the passage defined by A.

Perctl operator

The Perctl operator represents a percentile of a passage time density or distribution. Its evaluation yields:

$$\textit{Eval}(\text{Perctl}((\text{percentile}, \textit{subnode1}(\text{A})), (\text{density/distribution}, \textit{subnode2}(\text{B})))) \quad = \quad x$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1.

Eval(*subnode1*(A)) $= p$ is the percentile that is to be calculated (e.g. when seeking the 95^{th} percentile, $p = 95$).

If *subnode2* is of type PTD or Conv then *Eval*(*subnode2*(B)) $= f_B(t)$, where $f_B(t)$ is the probability density function of P_B, the random variable corresponding to the first passage time of the passage defined by B. Then, $F(x) = \int_0^x f_B(t)\,dt = \frac{p}{100}$, where $F(x)$ represents a probability value at time x – the time value that represents the p^{th} percentile of the probability distribution.

Alternatively, if *subnode2* is of type Dist or SS:P then *Eval*(*subnode2*(B)) $= F(t)$, where $F(t)$ is a probability distribution function. When *subnode2* is of type Dist, we have that $F(t) = \frac{p}{100}$, where $F(t)$ represents a probability value at time t, t being the time value that is the p^{th} percentile of the probability distribution. When calculating a percentile of a steady-state probability distribution, in case *subnode2* is of type SS:P, the distribution's non-continuous nature can result in the same time value being returned for a number of probability values (see Figure 4.1). Hence, we have $F(t) = p_1$ where $p_1 \leq F(t) \leq p_2$ and $p_1 \leq \frac{p}{100} \leq p_2$.

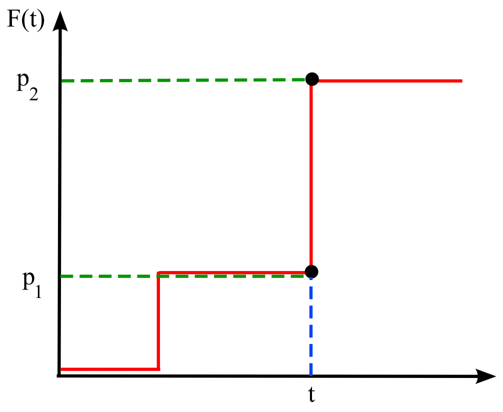


Figure 4.1: Illustration of how multiple probability values can map onto the same time value in steady-state distributions

Conv operator

The Conv operator represents the convolution of two passage time densities. Its evaluation yields:

$$\textit{Eval}(\text{Conv}((\text{density1}, \textit{subnode1}(\text{A})), (\text{density2}, \textit{subnode2}(\text{B})))) = \\ f \circ g = \int_0^t f(\tau)g(t-\tau)\, d\tau$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1, *Eval*(*subnode1*(A)) $= f(t)$, and *Eval*(*subnode2*(B)) $= g(t)$.

ProbInInterval operator

The ProbInInterval operator represents the probability with which a passage takes place in a certain amount of time. Its evaluation yields:

$$\textit{Eval}(\text{ProbInInterval}((\text{density}, \textit{subnode1}(\text{A})), (\text{time range}, \textit{subnode2}(\text{B})))) = \\ \int_{t_1}^{t_2} f(t)\, dt$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1, *Eval*(*subnode1*(A)) $= f(t)$, and *Eval*(*subnode2*(B)) $= [t_1, t_2]$.

ProbInStates operator

This operator represents the probability of a system occupying a set of states at a particular moment in time, having started in an initial state at time 0. Evaluation of the operator yields:

$$\textit{Eval}(\text{ProbInStates}((\text{observed states}, \textit{subnode1}(\text{A})), (\text{time instant}, \textit{subnode2}(\text{B})))) \quad = \\ \pi_{IJ}(t)$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1. We have $I = \{s\}$, where s is the initial state of the system , *Eval*(*subnode1* (A)) $= J$, and *Eval*(*subnode2*(B)) $= t$. π_{IJ} is the transient state distribution of the system, and is defined as $\pi_{IJ} = \mathbb{P}(\mathcal{X}(t) \in J \mid \mathcal{X}(0) \in I)$, where $\mathcal{X}(t)$ is the state of the system at time instant t.

Moment operator

The Moment operator represents a raw moment of a passage time density or distribution.

The n^{th} moment of a real-valued function $f(x)$ of a real variable about a value c is defined as:

$$\mu_n = \int_{-\infty}^{\infty} (x-c)^n f(x)\, dx$$

The moments about zero are usually referred to simply as the *raw moments* of a function. The n^{th} raw moment of a probability distribution function $f(x)$ is the expected value of the random variable X^n. The moments about its mean, μ, where

$$\mu_n = \int_{-\infty}^{\infty} (x-\mu)^n f(x)\, dx$$

are called *central moments*, and they describe the shape of the function. Raw moments can be used to calculate central moments. The first central moment is zero, the second central moment is the variance, and its square root is the standard deviation. Further measures, such as skewness and kurtosis, can be calculated from the third and fourth central moments.

Evaluation of the operator uses the Laplace transform method from [Dingle04a] and yields:

Eval(Moment((moment, *subnode1*(A)), (density/distribution, *subnode2*(B)))) $=$
$(-1)^n L^{(n)}(0)$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1. We have that *Eval*(*subnode1*(A)) $= n$, *Eval*(*subnode2*(B)) $= f(t)$ and $L(s) = L\{f(t)\}$ is the Laplace transform of $f(t)$.

FR operator

The FR operator represents the average rate of occurrence of a set of system actions. In the context of SPNs, this corresponds to the average firing rate of a transition. Evaluation of the operator yields:

$$\textit{Eval}(\text{FR}((\text{actions}, \textit{subnode}(\text{A})))) \quad = \quad \sum_{a \in B} \sum_{s:\text{enables } a} R_a(s)\pi_s$$

where *subnode* is a valid subnode of the operator, according to the syntax of Section 4.1, *Eval*(*subnode*(A)) = B, and $R_a(s)$ is the rate of occurrence of action a in state s.

SS:P operator

The SS:P operator represents the probability mass function yielding the steady-state probability of each possible value taken on by a StateFunc when evaluated over a given set of states.

Given a state function A, which associates a real value with every state of the system, the SS:P operator represents the steady-state probability distribution with which A takes on particular values.

The operator has a required and an optional subnode. The required subnode is an operator that represents state function A, and the optional subnode represents a set of states that are to be considered. If the optional subnode is not provided, the steady-state distribution is calculated over all the states of the system. If, however, it is provided, the distribution is calculated only over the set of states represented by the optional subnode. Evaluation of the operator yields:

$$\textit{Eval}(\text{SS:P}((\text{state function}, \textit{subnode1}(\text{A})), (\text{states}, \textit{subnode2}(\text{B})))) \quad = \quad \mathbb{P}(Z_A = r)$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1. Z_A represents the value of state function A on a state, and

$$\mathbb{P}(Z_{\mathrm{A}} = r) = \begin{cases} \displaystyle\sum_{\substack{s\,:\,s\,\models \mathrm{B},\\ \mathit{sfEval}(\mathrm{A},s)=r}} \pi_s & \text{iff } r \in \{\mathit{sfEval}(\mathrm{A}, s) : s \models \mathrm{B}\} \\ 0 & \text{otherwise} \end{cases}$$

π_s is the steady-state probability of state s. If the optional subnode has not been provided, B represents the set of all states of the model, i.e. $\mathrm{B} = \mathcal{S}$. *sfEval*(A, s) is defined by the semantics of the StateFunc operator.

SS:S operator

The SS:S operator represents the set of states that have a certain steady-state probability. Its evaluation yields:

$$\mathit{Eval}(\text{SS:S}((\text{prob. range}, \mathit{subnode1}(\mathrm{A})))) = \Big\{ s : \mathcal{S} \mid p_1 \leq \pi_s \leq p_2 \Big\}$$

where *subnode1* is a valid subnode of the operator, according to the syntax of Section 4.1; *Eval*(*subnode1*(A)) = $[p_1, p_2]$ and π_s is the steady-state probability of state s. Due to our assumption that models have irreducible state spaces, the specification of the set of start states is not required, as these have no impact on the result.

StatesAtTime operator

The StatesAtTime operator represents the set of states that the system can occupy at a given time instant with a certain probability. The evaluation of the operator yields:

$$\mathit{Eval}(\text{StatesAtTime}((\text{time instant}, \mathit{subnode1}(\mathrm{A})), (\text{prob. range}, \mathit{subnode2}(\mathrm{B})))) = \\ \Big\{ s : J \ \mid \ p_1 \leq \pi_{IJ}(t) \leq p_2 \Big\}$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1,

- *Eval*(*subnode1*(A)) $= t$,

- *Eval*(*subnode2*(B)) $= [p_1, p_2]$,

and π_{IJ} is the transient state distribution of the system, and is defined as $\pi_{IJ} = \mathbb{P}(\chi(t) \in J \mid \chi(0) \in I)$, where $I = \{s\}$, with s being the initial state of the system at time 0, and $\chi(t)$ being the state of the system at time instant t.

InInterval operator

The InInterval operator represents a Boolean value that determines whether a numerical value lies within a given interval. Evaluation yields:

$$\textit{Eval}(\text{InInterval}((\text{num. value}, \textit{subnode1}(\text{A})), (\text{range}, \textit{subnode2}(\text{B})))) = \begin{cases} \textit{true} & \text{iff } r_1 \leq n \leq r_2 \\ \textit{false} & \text{otherwise} \end{cases}$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1, and we also have that *Eval*(*subnode1*(A)) $= n$, and *Eval*(*subnode2*(B)) $= [r_1, r_2]$.

Macro operator

The Macro operator defines custom performance concepts as compositions of other operators.

In this way, it is able to represent a large number of diverse user-defined concepts. Its semantics are similar to that of the ? operator, since the Macro operator represents an entire hierarchy of operators. Thus, its evaluation yields:

$$\textit{Eval}(\text{Macro}((\text{macro}, \textit{subnode}(\text{A})))) \quad = \quad \textit{Eval}(\textit{subnode}(\text{A}))$$

where *subnode* is the topmost operator of the hierarchy represented by the Macro operator. Evaluation of the operator takes place recursively.

$\subseteq$ **operator**

The $\subseteq$ operator represents a Boolean value that determines whether a set is a subset of another set. Its evaluation yields:

$$\textit{Eval}(\subseteq((\text{set 1}, \textit{subnode1}(\text{A})), (\text{set 2}, \textit{subnode2}(\text{B})))) \quad = \quad \begin{cases} \textit{true} & \text{iff } s_1 \subseteq s_2 \\ \textit{false} & \text{otherwise} \end{cases}$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1. We have that *Eval*(*subnode1*(A)) $= s_1$, and *Eval*(*subnode2*(B)) $= s_2$.

$\vee/\wedge$ **operator**

The $\vee/\wedge$ operator represents the logical disjunction or conjunction of two Boolean values.

For the case when it represents a logical disjunction, its evaluation yields:

$$\textit{Eval}(\vee((\text{bool value 1}, \textit{subnode1}(\text{A})), (\text{bool value 2}, \textit{subnode2}(\text{B})))) = \begin{cases} \textit{true} & \text{iff } b_1 == \textit{true} \text{ or } b_2 == \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}$$

For the case when it represents a logical conjunction, its evaluation yields:

$$\textit{Eval}(\wedge((\text{bool value 1}, \textit{subnode1}(\text{A})), (\text{bool value 2}, \textit{subnode2}(\text{B})))) = \begin{cases} \textit{true} & \text{iff } b_1 == \textit{true} \text{ and } b_2 == \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1, and in both cases *Eval*(*subnode1*(A)) $== b_1$, and *Eval*(*subnode2*(B)) $== b_2$.

$\neg$ **operator**

The $\neg$ operator represents the logical negation of a Boolean value. Its evaluation yields:

$$\textit{Eval}(\neg((\text{bool value}, \textit{subnode}(\text{A})))) = \begin{cases} \textit{true} & \text{iff } b = \textit{false} \\ \textit{false} & \text{otherwise} \end{cases}$$

where *subnode* is a valid subnode of the operator, according to the syntax of Section 4.1, and *Eval*(*subnode*(A)) $= b$.

$\bowtie$ operator

The $\bowtie$ operator represents a Boolean value that is the result of the arithmetic comparison of two numerical values with standard numerical comparison operators. Its evaluation for the various cases yields:

$$\textit{Eval}(\bowtie((\text{num. value 1}, \textit{subnode1}(\text{A})), (\text{num. value 2}, \textit{subnode2}(\text{B})))) = \begin{cases} \textit{true} & \text{iff } n_1 \bowtie n_2 \\ \textit{false} & \text{otherwise} \end{cases}$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1. We also have that *Eval*(*subnode1*(A)) $= n_1$, *Eval*(*subnode2*(B)) $= n_2$, and $\bowtie \; \in \{<, \leq, ==, \geq, >\}$.

$\oplus$ operator

The $\oplus$ operator represents a numerical value that is the result of an arithmetic operation on two numerical values, using standard arithmetic operators. Its evaluation for the various cases yields:

$$\textit{Eval}(\oplus((\text{num. value 1}, \textit{subnode1}(\text{A})), (\text{num. value 2}, \textit{subnode2}(\text{B})))) = n_1 \oplus n_2$$

where *subnode1* and *subnode2* are valid sub-nodes of the operator, according to the syntax of Section 4.1. We also have that *Eval*(*subnode1*(A)) $= n_1$, *Eval*(*subnode2*(B)) $= n_2$, and $\oplus \in \{+, -, *, /, \hat{}\}$.

Chapter 5

Tool Support for Performance Trees

This chapter introduces tool support for Performance Tree-based query specification and evaluation by describing a prototype performance analysis environment that enables the graphical creation and parallel and distributed evaluation of GSPN-based system models and Performance Tree queries. We introduce *PIPE2*, a platform-independent open-source Petri net tool, which serves as the graphical front-end to this environment. We discuss how *PIPE2* facilitates the convenient creation of GSPN-based system models, and how we implement support for the interactive graphical and natural language-based specification of Performance Tree queries in a new analysis module. We also describe a range of state-of-the-art parallel and distributed performance analysis tools, which supply performance query evaluation capabilities to the analysis environment. We conclude with a discussion of how we distribute performance queries to applicable analysis tools for evaluation on a dedicated Grid-based analysis cluster. Material from [Brien08b, Suto08a, Suto08b, Bradley08, Dingle08b] is incorporated into this chapter.

5.1 *PIPE2*: A Tool for GSPN-based System Modelling and Analysis

PIPE2 [PIPE, Bonet07, Suto08b] is a Java-based open-source tool for stochastic system modelling and performance analysis. It was originally developed as a platform-independent Petri net editor to support the creation and manipulation of potentially complex GSPN models through a simple and intuitive graphical user interface. Subsequently, it has been further enhanced by the integration of a number of analysis modules, and has as a result of our work evolved into a versatile front-end for an integrated parallel and distributed performance evaluation environment. With *PIPE2*, users are now able to design system models, visually create complex queries that address models in terms of performance properties of interest, and evaluate them to obtain relevant performance metrics or to establish the validity of certain performance properties.

5.1.1 Model Editor

PIPE2 provides a graphical user interface (shown in Figure 5.1) that allows the creation, editing, saving and loading of GSPN models that conform to the Petri Net Mark-up Language (PNML) [PNML] document interchange format. PNML enables *PIPE2* to import and manipulate models created externally, and allows other PNML-based tools to do so with models created in *PIPE2*.

Models are drawn on a canvas using features from a drawing toolbar. GSPN models are constructed from graphical components representing places, transitions, arcs and tokens. Nets of arbitrary complexity can be drawn and annotated with additional user information. Besides basic model design functionality, the designer interface also provides additional visual features, such as zoom, export, tabbed editing and animation. The animation mode is particularly useful for aiding users in the intuitive verification of the behaviour of their models, since it enables them to visualise models in action.

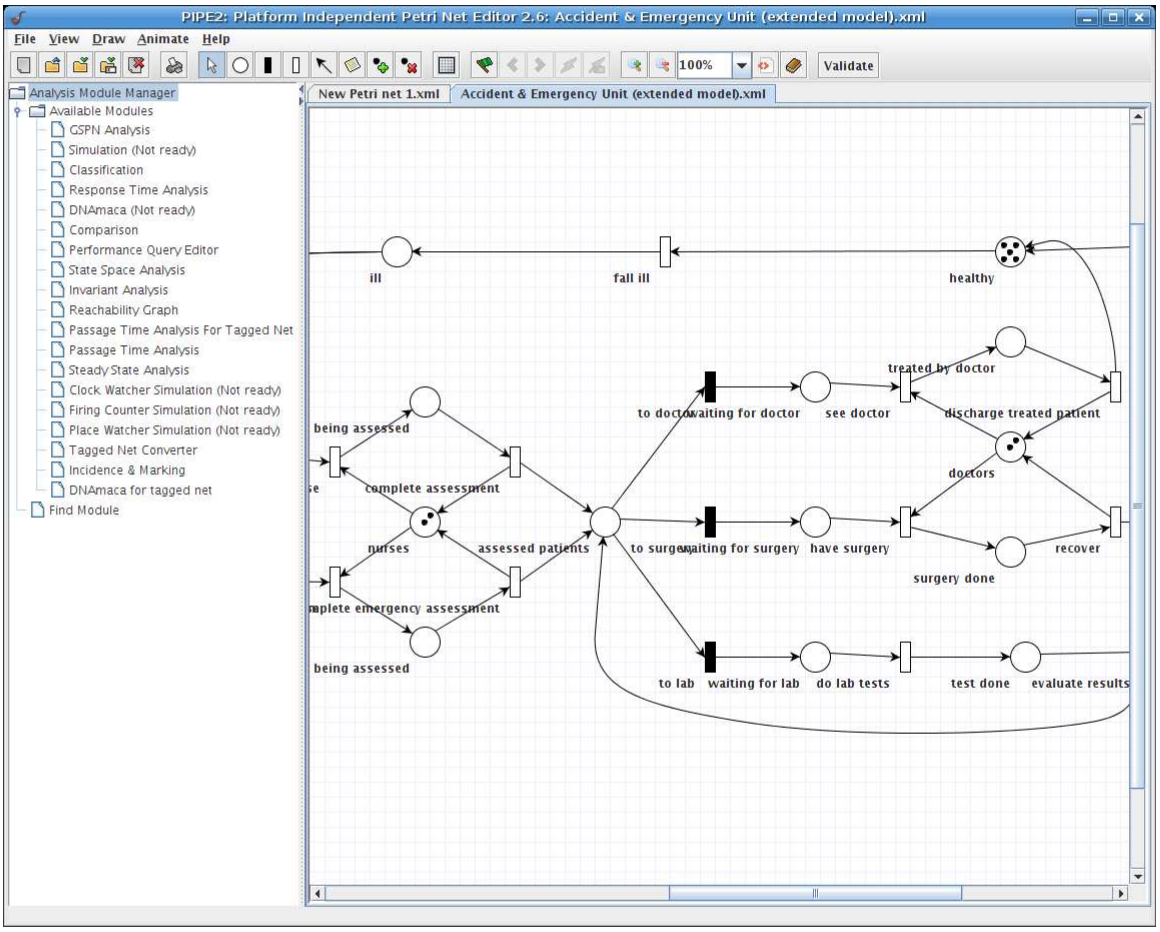


Figure 5.1: GSPN Model Designer Interface

5.1.2 Analysis Modules

PIPE2 is equipped with a number of specialised analysis modules that perform structural and performance-related analyses on GSPN models. A panel to the left of the canvas enables users access to the currently available modules. Structural analyses examine models in terms of their topology, and are able to verify whether some aspects of their qualitative behaviour are in accordance with expectations derived from the real-life systems that are to be modelled. Performance analyses are aimed at investigating models from their operational point of view, and are therefore able to provide deeper insights into quantitative aspects of their behaviour. At present, modules exist for the classification and comparison of SPN models, for the derivation of their reachability graphs, for the analysis of their state spaces, invariants, incidence matrices and markings, and for simulation and interfacing with the *DNAmaca* tool (see Section 2.4.1).

Structural Analysis Modules

Model Classification Module: Classifies GSPN models based on their structure into the following categories: state machine, marked graph, free choice net, extended free choice net, simple net and extended simple net.

Model Comparison Module: Compares two GSPN models based on attributes determined by users as comparison criteria.

State Space Module: Constructs a graph of all reachable states, which is used to determine properties of GSPN models, such as liveness, boundedness and existence of deadlocks.

Incidence & Marking Module: Determines and displays the forward and backward incidence matrices, the marking matrix, and the set of enabled transitions of the GSPN model.

Reachability Graph Module: Provides a visual representation of all possible transition firing sequences of the GSPN model, and informs users about possible states that the model can enter.

Performance Analysis Modules

Simulation Module: Studies the performance of models by investigating the average number of tokens per place, using a Monte Carlo simulation-based approach.

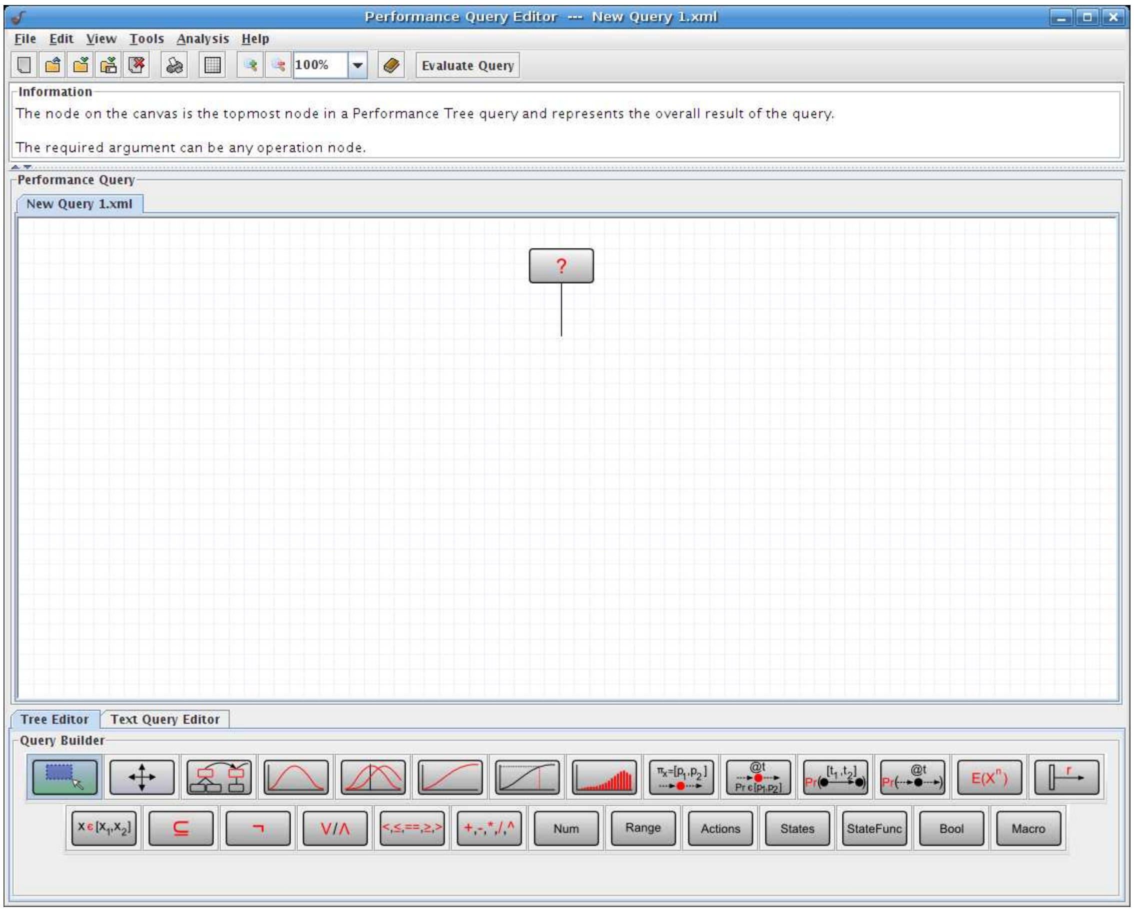
Steady-State Analysis Module: Calculates state and count measures from the steady-state distribution via an interface to the *DNAmaca* steady state analyser.

Passage Time Analysis Module: Calculates probability densities for the time taken for a system represented by a model to complete a user-defined passage via an interface to the *SMARTA* passage time analyser.

GSPN Analysis Module: Calculates analytically the average number of tokens on places, token density and the throughput of timed transitions.

5.1.3 Performance Query Editor

As part of our present work, *PIPE2* has been extended with the *Performance Query Editor (PQE)* module [Brien08b, Suto08b, Dingle08b], which allows the graphical design and subsequent parallel and distributed evaluation of complex performance queries, expressed as Performance Trees. It implements an easy-to-use graphical user interface for the construction of performance queries, which is shown in Figure 5.2.



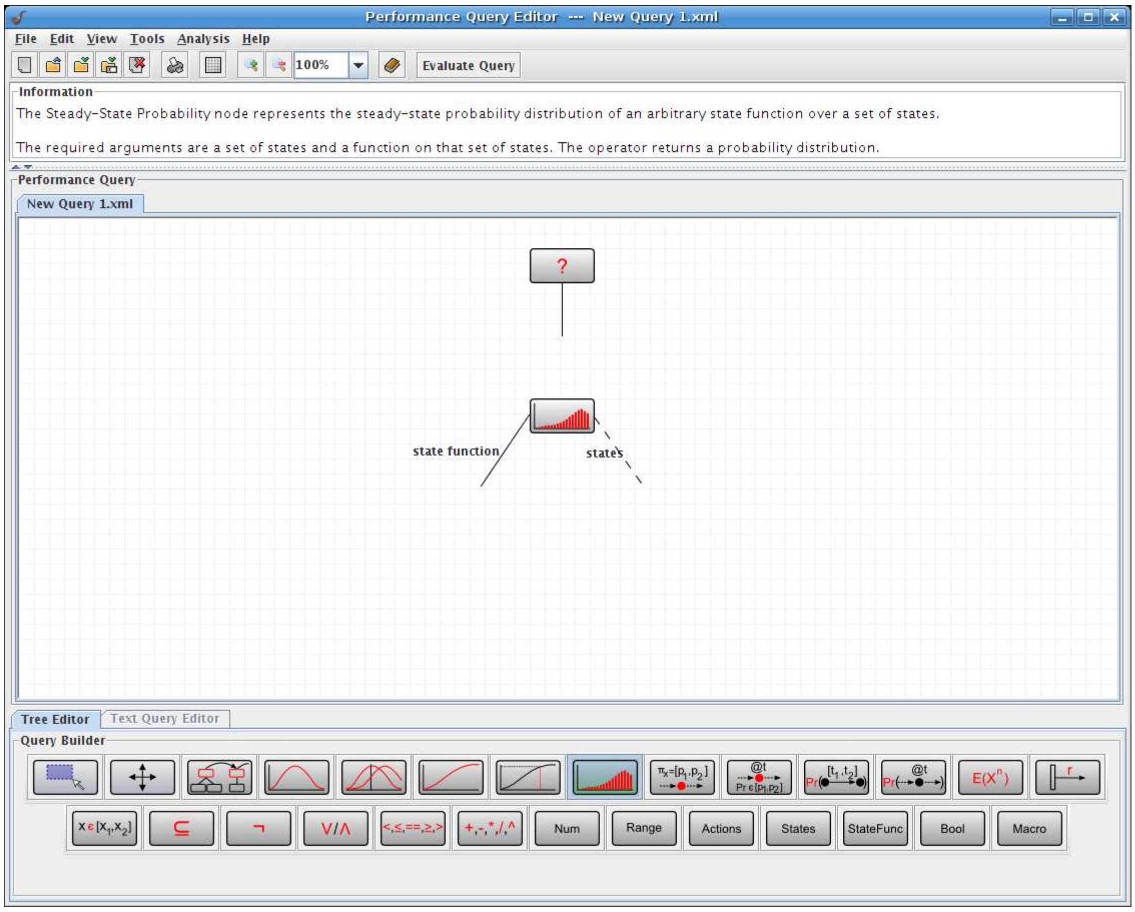
The node on the canvas is the topmost node in a Performance Tree query and represents the overall result of the query.

The required argument can be any operation node.

Figure 5.2: GUI of the Performance Query Editor module

Graphical Query Specification

Like the model editor, the largest part of the PQE's GUI is occupied by the canvas, the drawing area that serves as a container for the graphical components of a performance query. Users construct Performance Tree queries on the canvas using the functionality of the query designer toolbar, which is located below the canvas. When a toolbar button representing a Performance Tree node is selected, it becomes highlighted, indicating that an instance of the node it represents is to be drawn onto the canvas at the location of the mouse click. This is illustrated in Figure 5.3.


Information

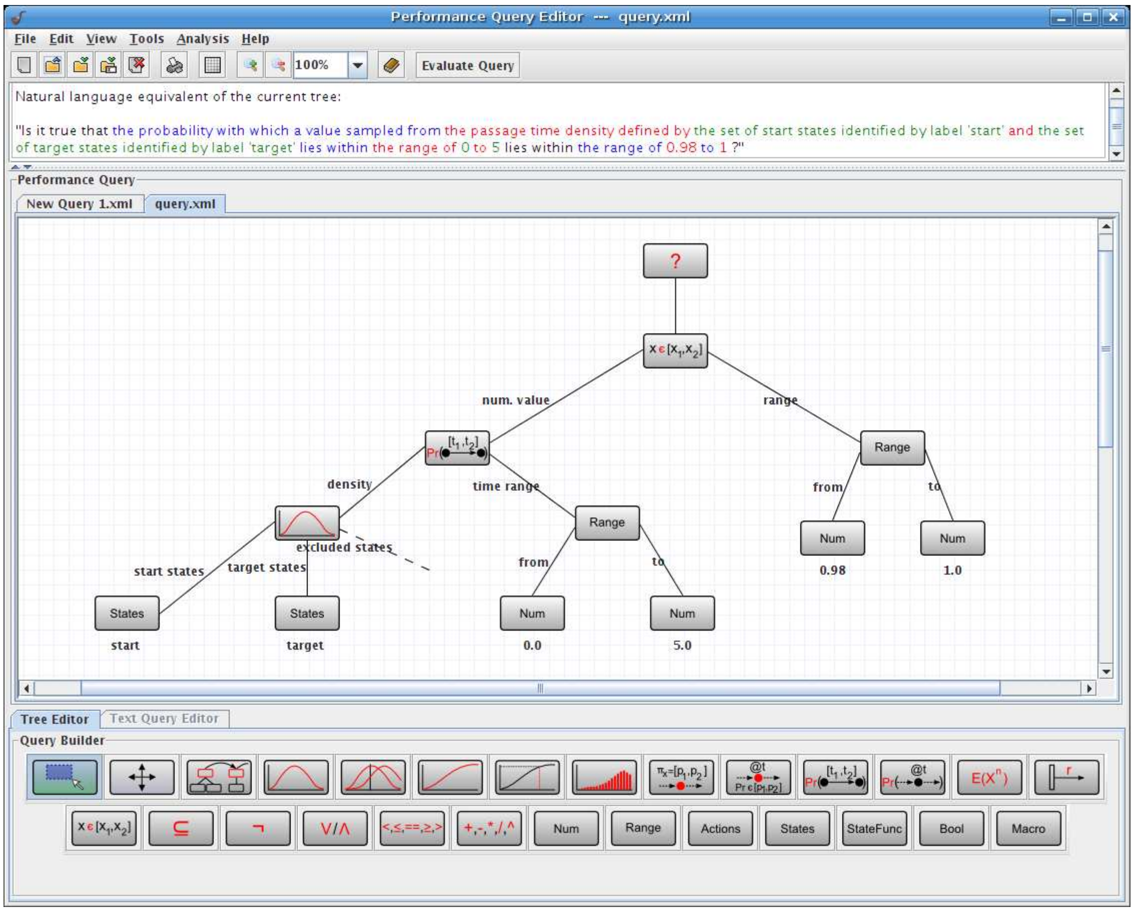
The Steady-State Probability node represents the steady-state probability distribution of an arbitrary state function over a set of states.

The required arguments are a set of states and a function on that set of states. The operator returns a probability distribution.

Figure 5.3: PQE GUI showing a newly drawn operation node

When an operation node is drawn onto the canvas, a number of arcs are also created, which emanate from the bottom of the node. Arcs either appear as solid or dashed lines, and are annotated by labels. A solid arc indicates that a required sub-node is to be provided to the operation node via the arc connection. A dashed arc indicates that an optional sub-node can be provided to the operation node in addition by connecting it to the arc. An arc label represents the role that the sub-node connected to the arc has for the operation node. In the case of Figure 5.3, the node that will be connected to the arc with the label ‘state function’ will be considered by the SS:P node to represent a state function. As a further illustration of the concept, consider the left-bottommost States node in Figure 5.4, which is connected to an arc with label ‘start states’. This connection indicates that its PTD parent node considers the states represented by the States node to define the start states of the passage.

Queries can be rearranged at will, as each individual graphical component on the canvas can be manipulated independently. Nodes can be repositioned, disconnected from arcs or deleted. The PQE is initialised with a canvas that by default contains a single Result node with an outgoing arc, as illustrated by Figure 5.2. Queries are constructed by linking nodes together, with the hierarchy eventually terminating in the Result node. Each query has a single Result node. Figure 5.4 shows a fully constructed Performance Tree query on the PQE GUI.



Natural language equivalent of the current tree:

"Is it true that the probability with which a value sampled from the passage time density defined by the set of start states identified by label 'start' and the set of target states identified by label 'target' lies within the range of 0 to 5 lies within the range of 0.98 to 1 ?"

Figure 5.4: GUI showing a fully constructed Performance Tree query

To provide additional support for query specification, the PQE module also incorporates an automatic query interpretation mechanism that translates Performance Tree queries into their natural language equivalent on-the-fly, as queries are being constructed. This is a particularly useful feature for users of the tool who are not fully accustomed to Performance Trees. The natural language representation enables them to intuitively verify whether the performance query that they are constructing corresponds to the query that they have originally intended to express. On the GUI, a pane located above the canvas displays the query's natural language equivalent. This pane also serves as a general information interface, which provides users with usage instructions for individual Performance Tree nodes when they are selected. The natural language representation is colour-encoded to represent the hierarchy within the query, which is immediately apparent in Performance Tree form.

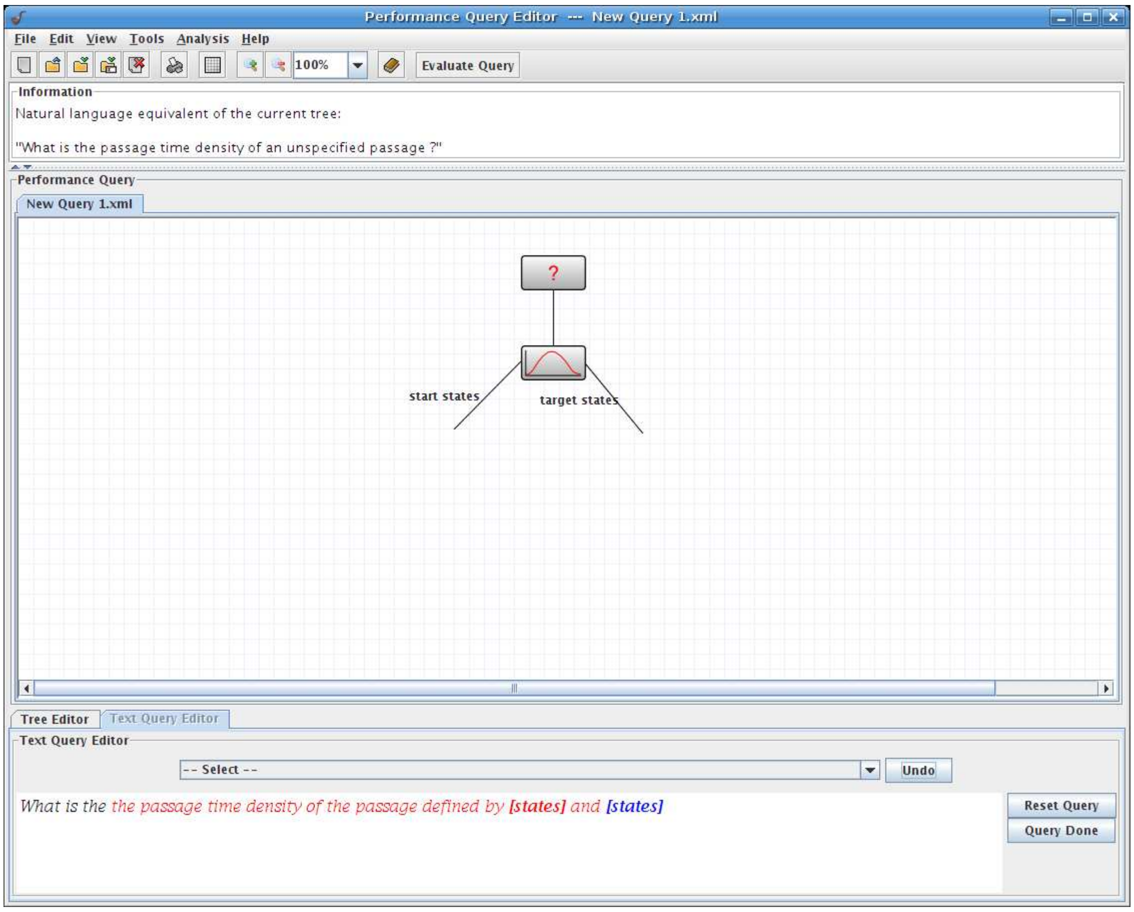
To ensure *valid* performance queries, on-the-fly type compatibility validation is carried out during the construction of Performance Tree queries whenever node assignments are attempted. When users want to assign a node as a sub-node to an operation node, they attempt to connect the operation node's outgoing arc to the node. At this point, a comparison between the type of the node and the types of acceptable sub-nodes is performed. If the node's type forms part of the set of acceptable sub-node types, the node assignment is considered valid and the arc connection is established. If it does not, the node cannot be assigned to the operation node.

To ensure *complete* performance queries, a complete structural validation of a performance query is effected when users have finished its construction and have requested its evaluation. At this point, it may be the case that a structurally valid Performance Tree has been constructed, but that certain required arguments have not been specified. Thus, a validation mechanism needs to verify whether a valid tree hierarchy exists, whether all operation nodes' required sub-nodes have been provided, and whether value nodes have been properly defined. Only when all of these validation criteria have been satisfied do performance queries proceed to evaluation.

Natural Language-based Query Specification

The construction of Performance Tree queries becomes routine once users are accustomed to the hierarchical tree structure-based representation and have become comfortable with the concepts that Performance Tree nodes represent. Nevertheless, it may initially be more convenient and intuitive to new users to specify performance queries in natural language.

[Grunske08a] and [Grunske08b] introduce the idea of using a pattern system for common probabilistic properties, together with a structured English grammar for aiding users in the query specification process. We have devised a similar approach and implemented a guided query specification mechanism in the PQE module that is based on a structured grammar and that allows Performance Tree queries to be constructed incrementally in natural language [Wang08]. Figure 5.5 shows the natural language query specification interface located below the canvas. Here, a drop-down menu and a text area take the place of the query builder toolbar, which is provided to users as the default option for Performance Tree query construction. The drop-down menu is the main point of interaction for users, and is updated dynamically to continuously offer a selection of currently valid expressions that can be used to further extend the partially constructed query. Once a selection has been made, the Performance Tree query is updated accordingly.



Natural language equivalent of the current tree:

"What is the passage time density of an unspecified passage ?"

What is the the passage time density of the passage defined by ***[states]*** *and* ***[states]***

Figure 5.5: Natural language-based performance query construction

This mechanism provides users with the comfort of a very simple and straightforward query specification mechanism. At the same time, due to the automatic construction of Performance Trees that correspond to the specified natural language queries, users are aided in their familiarisation with the direct graphical specification of Performance Tree queries.

5.2 An Integrated Evaluation Environment for Performance Trees

To provide users with a complete toolset that is able to design system models, specify performance queries on them in the form of Performance Trees, and evaluate these in order to obtain relevant results, we have developed a sophisticated performance analysis environment that supports all of this functionality in an integrated manner.

Our analysis environment consists of a number of interacting software components. *PIPE2* serves as a graphical front-end to users, which allows them to carry out model design, performance query specification and evaluation tasks in a convenient manner. For query evaluation purposes, *PIPE2* interfaces with an analysis server, which collectively coordinates a number of parallel and distributed analysis tools, each of which specialises in different types of analyses. From a user's perspective, all analysis functionality is handled by *PIPE2*, and the seamlessness of its integration with the analysis environment creates the illusion of all analysis functionality being a feature of the tool. Computations are carried out by the analysis tools on a dedicated analysis cluster, which is configured as a Grid resource. The advantage of using a Grid cluster for analysis purposes is that it can incorporate a large number of heterogeneous resources – thereby enabling it to support large-scale computations – and that it can be easily extended. Most importantly, however, a Grid cluster is administered by sophisticated middleware, which provides a complete suite of cluster and job management capability.

At present, we are able to analyse models whose size does not exceed $O(10^8)$ states. Overall evaluation capacity is ultimately determined and hence constrained by the individual capacities of the analysis tools that implement evaluation support. As tools with enhanced solution capacity are introduced into the environment, its overall evaluation capacity increases accordingly.

Figure 5.6 illustrates the individual components of the analysis environment and their interactions. The analysis environment consists of two main parts. One part encompasses client-side components that users are directly exposed to and that are implemented in *PIPE2*, and another part contains server-side components that perform analysis tasks and that users are not directly aware of. These include the Analysis Server, which deals with requests from *PIPE2* and interacts with analysis tools, the analysis tools themselves, which carry out a range of specialised analyses, and the analysis cluster, which provides the underlying computational hardware infrastructure.

5.2.1 Analysis Client

PIPE2 has the role of the client within the analysis environment, and is its only user-facing component. It is the gateway to the functionality provided by the analysis environment's other components, and allows users to create system models and applicable Performance Tree queries. It initiates query evaluation by communicating model and query data to the Analysis Server for further processing. As soon as the Analysis Server has obtained evaluation results, *PIPE2* obtains and presents them to users in a visually accessible manner.

Figure 5.7 illustrates how users can initiate the evaluation of a fully constructed Performance Tree query by clicking on the 'Evaluate Query' button on the GUI. Figure 5.8 depicts the performance query evaluation tracker interface, which shows a replica of the query tree, in which every node is annotated with a status indicator that follows a conventional traffic light colouring scheme. On initial submission, the status indicator of every node of the tree is red, indicating that the nodes have not been scheduled for evaluation yet. Once evaluation has commenced, the indicators pulse in yellow, signalling that their nodes have been submitted to the analysis tools for processing. As results for nodes are obtained by *PIPE2*, the status indicators of the respective nodes turn green. This indicates to users that they can click on nodes to visualise results. This is possible while other eval-

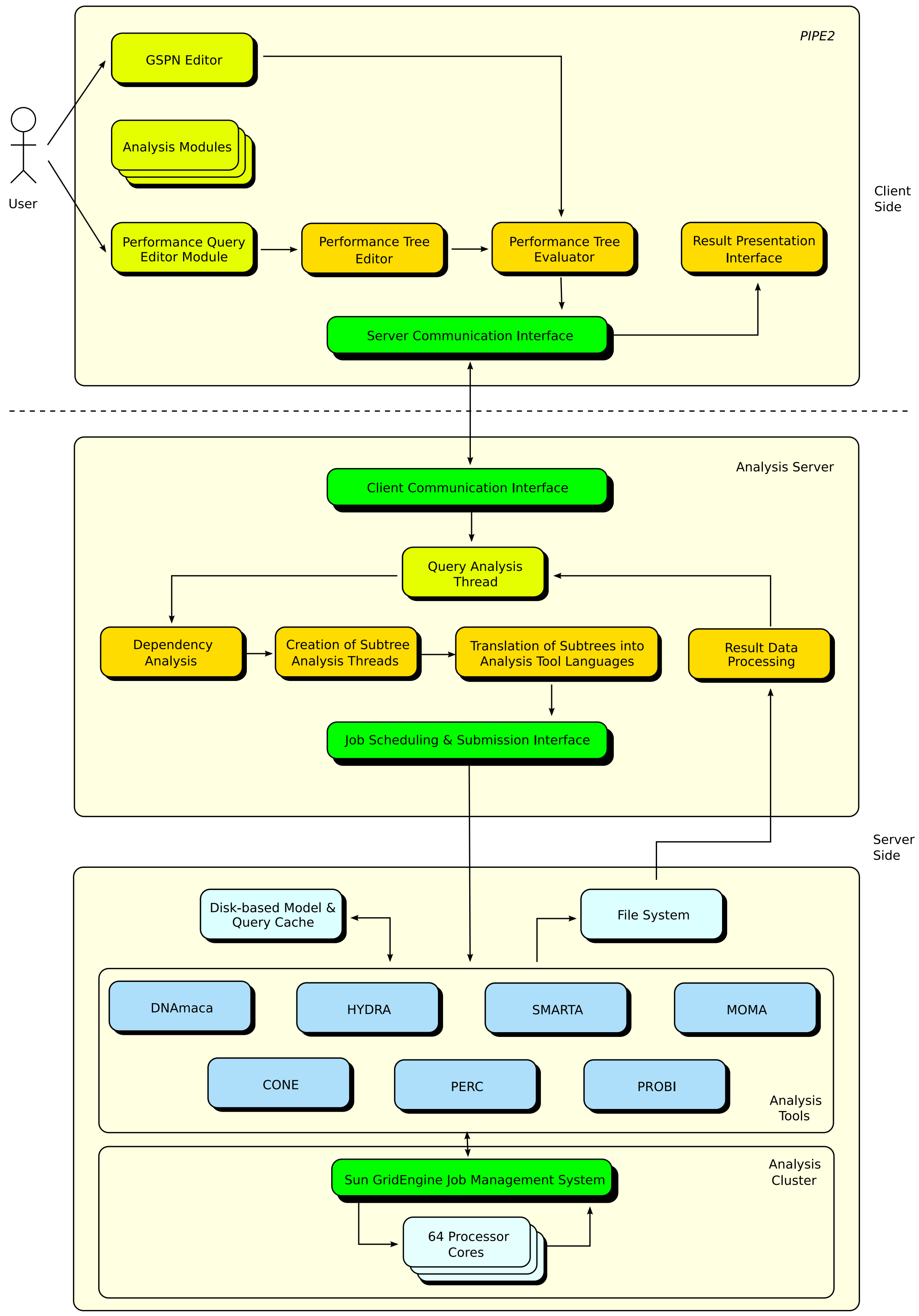
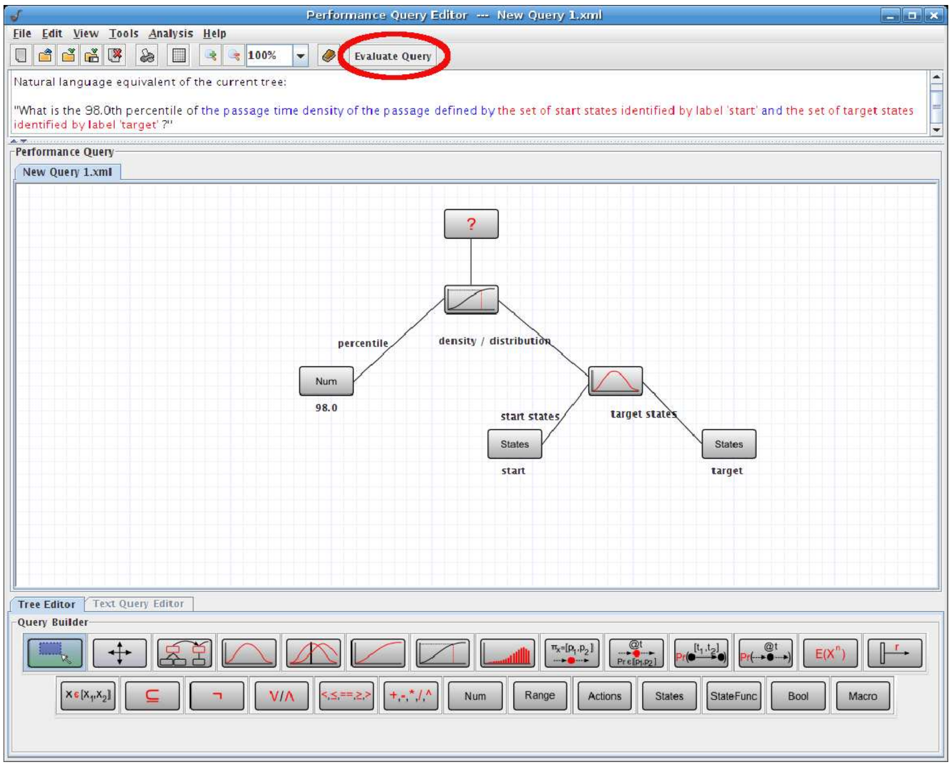

PIPE2
GSPN Editor
Analysis Modules
User
Performance Query Editor Module
Performance Tree Editor
Performance Tree Evaluator
Result Presentation Interface
Server Communication Interface
Client Side
Analysis Server
Client Communication Interface
Query Analysis Thread
Dependency Analysis
Creation of Subtree Analysis Threads
Translation of Subtrees into Analysis Tool Languages
Result Data Processing
Job Scheduling & Submission Interface
Server Side
Disk-based Model & Query Cache
File System
DNAmaca
HYDRA
SMARTA
MOMA
CONE
PERC
PROBI
Analysis Tools
Analysis Cluster
Sun GridEngine Job Management System
64 Processor Cores

Figure 5.6: Performance analysis environment architecture

uations are still in progress. Using an example of a percentile of a passage time density, Figure 5.9 shows how graph-based results are visualised by *PIPE2*.



Natural language equivalent of the current tree:

"What is the 98.0th percentile of the passage time density of the passage defined by the set of start states identified by label 'start' and the set of target states identified by label 'target' ?"

Figure 5.7: Initiation of a performance query evaluation request

5.2.2 Analysis Server

The Analysis Server is responsible for the processing of evaluation requests issued by *PIPE2*, and the coordination of the subsequent performance query evaluation process. It is deployed at the analysis cluster's primary host, and is continuously available to accept incoming analysis requests. The Analysis Server processes these requests by decomposing performance queries into subtrees and sending these to specialised analysis tools for evaluation. The Analysis Server is also responsible for passing on evaluation results to *PIPE2* for visualisation.

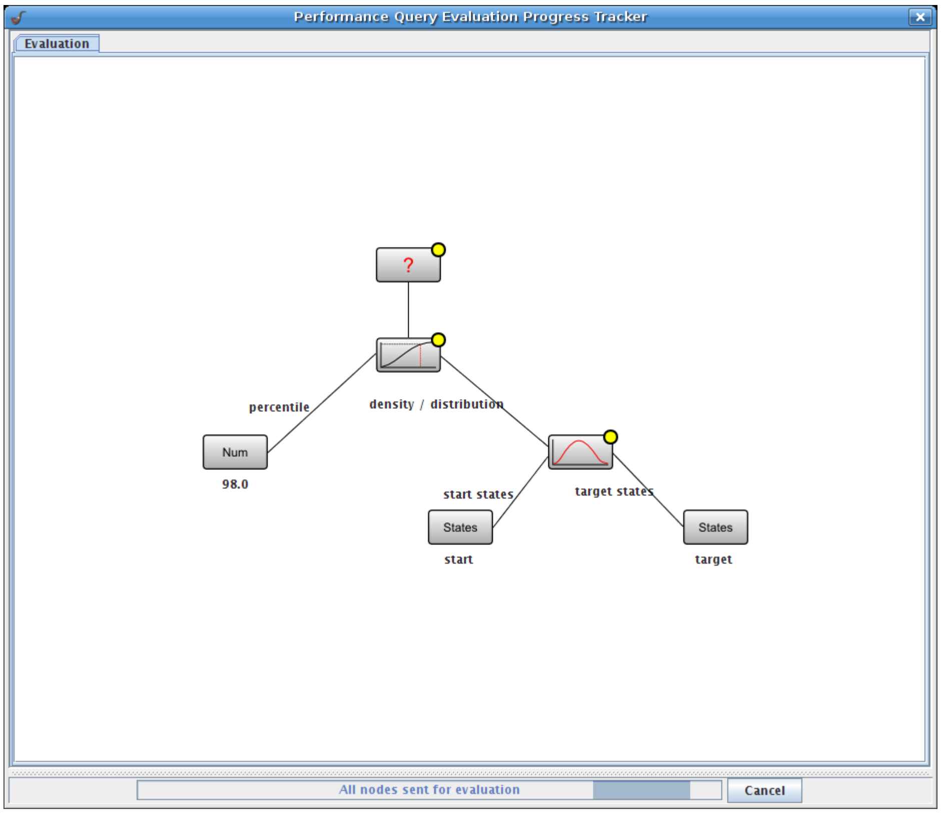


Figure 5.8: Query evaluation progress tracker interface

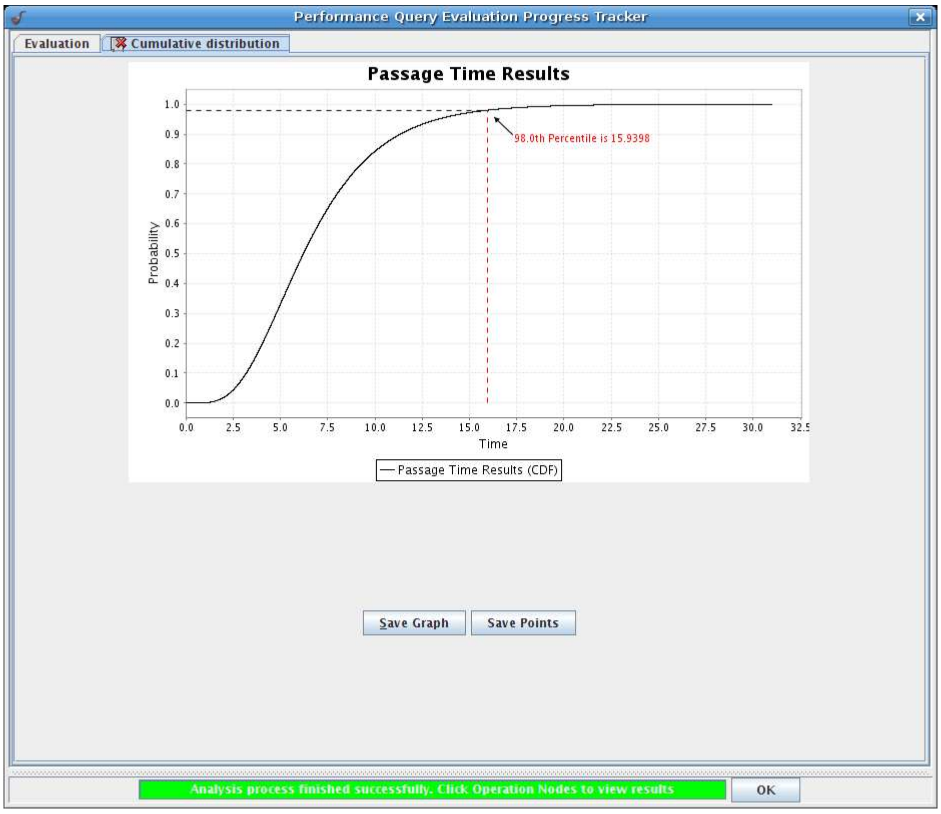

Performance Query Evaluation Progress Tracker
Evaluation
Cumulative distribution
Passage Time Results
98.0th Percentile is 15.9398
Probability
Time
Passage Time Results (CDF)
Save Graph
Save Points
Analysis process finished successfully. Click Operation Nodes to view results
OK

Figure 5.9: Visualisation of performance query results

5.2.3 Analysis Tools

The evaluation of quantitative measures defined in Performance Tree queries is ultimately carried out by a set of analysis tools that are invoked by the Analysis Server. At present, tools are available for the calculation of steady-state and transient measures, passage time densities and distributions, as well as their convolutions, moments and percentiles. It is the integration of these analysers with the evaluation environment that enables the evaluation of Performance Tree nodes. Tools currently forming part of the analysis environment are:

DNAmaca [Knottenbelt96] – a Markov chain steady-state analyser that can solve models with up to $O(10^8)$ states. It supports model and performance measure specification in its proprietary input language, performs functional and steady-state analyses, and computes performance statistics, such as the mean, variance and standard deviation of expressions computed on system states. In addition, it also calculates mean rates of occurrence of actions. The raw distribution from which these performance statistics are calculated can also be obtained. *DNAmaca* is used for the evaluation of the SS:P and FR Performance Tree operators.

SMARTA [Dingle04a] – a distributed MPI-based semi-Markov response time analyser that performs iterative numerical analyses of passage times in very large semi-Markov models (including GSPNs), using hypergraph partitioning and numerical Laplace transform inversion techniques. *SMARTA* is suitable for the analysis of the PTD and Dist operators on GSPN models where start and target states are vanishing.

HYDRA [Dingle04a] – a distributed Markovian passage time analyser that uses hypergraph partitioning and uniformisation techniques. *HYDRA* is suitable for the evaluation of the PTD and Dist Performance Tree operators, and also features transient analysis capabilities that are useful for the evaluation of the ProbInStates and StatesAtTime operators.

MOMA [Brien08a] – an n^{th} order raw moment calculator for GSPN models that uses a Laplace transform-based method. *MOMA* is used for the evaluation of the Moment Performance Tree operator.

CONE [Brien08a] – a performance analyser that, together with *SMARTA*, evaluates the convolution of two passage time densities using a Laplace transform and Laguerre inversion-based approach. *CONE* is used for the evaluation of the Conv Performance Tree operator.

PERC [Brien08a] – a performance analyser that calculates percentiles of passage time distributions and densities, and works in conjunction with *SMARTA*. *PERC* is used for the evaluation of the Perctl Performance Tree operator.

PROBI [Brien08a] – a performance analyser that calculates the probability of a passage time lying within a certain interval, and borrows functionality from *SMARTA*. *PROBI* is used for the evaluation of the ProbInInterval Performance Tree operator.

For more details on the individual performance analysis tools, consult Section 2.4.1.

5.2.4 Analysis Cluster

Camelot, the computational cluster forming the backbone of the analysis environment, consists of 16 dual-processor dual-core nodes, each of which is a Sun Fire x4100 with two 64-bit Opteron 275 processors and 8GB of RAM. Nodes are connected with both Gigabit Ethernet and Infiniband interfaces. The Infiniband fabric runs at 2.5Gbit/s, and is managed by a Silverstorm 9024 switch. Job submission is handled by Sun GridEngine (SGE), a Grid management middleware that configures and exposes *Camelot* as a computational Grid resource. Clients submit sequential and parallel (MPI) jobs to SGE via the Distributed Resource Management Application API (DRMAA).

5.3 Parallel and Distributed Evaluation of Performance Queries

Users interact with *PIPE2* to design system models and performance queries. The tool also enables them to initiate the automatic evaluation of performance queries through the single click of a button, and provides them with a convenient evaluation progress tracking and visual result feedback mechanism.

When a user requests a query's evaluation, *PIPE2* establishes a connection with the Analysis Server in the background, which resides on the analysis cluster's primary host and is responsible for the coordination of the distributed evaluation of performance queries. The Analysis Server delegates the processing of individual queries to dedicated analysis threads, in order to be able to serve multiple simultaneous requests without delay.

Analysis threads process serialised versions of system models and performance queries that have been submitted to the Analysis Server by *PIPE2*. They construct an internal representation of the data, based on input formats required by the analysis tools that need to be invoked for the evaluation of query nodes. All of our tools use the *DNAmaca* input language (see Section 2.2.4), together with tool-specific extensions. Hence, models are translated into *DNAmaca* model files, which are then extended with tool-specific syntax that specifies performance criteria according to which evaluation is to be carried out. Analysis threads decompose queries into a set of subtrees, each of which consists of a single query node. They subsequently create individual helper threads for each subtree to initiate and coordinate their evaluation. Once a helper thread has been created, it submits the subtree that it is responsible to process for evaluation in the form of an analysis job to SGE. Analysis jobs consist of analysis tool invocation requests. As already discussed in Section 5.2.3, different analysis tools are invoked on the analysis cluster, based on the type of subtrees. Threads communicate with SGE via a DRMAA interface. SGE has built-in scheduling algorithms that are used to distribute jobs onto available processors on the analysis cluster.

Once evaluation jobs have been scheduled on the analysis cluster and analysis tools have been provided with the required input data, evaluation commences. Analysis tools parse the supplied model files that also contain performance evaluation criteria, and compute the requested performance measures. In case the evaluation of a subtree is conditional on results to be obtained from the evaluation of other subtrees, the job is suspended until all required inputs have become available. Performance Tree queries have the advantage that they can be evaluated in a distributed fashion. During evaluation, parallelism takes place on two levels. Firstly, certain tools are able to carry out the evaluation of Performance Tree nodes in a parallelised manner. Secondly, if two nodes within the query tree are independent of one another, they can be evaluated by tools in parallel.

To avoid redundant work and thus reduce response time, the Analysis Server incorporates a disk-based caching mechanism that stores performance query evaluation results, so that subsequent evaluation requests for already evaluated queries on the same model can be served with results immediately. In order to differentiate between multiple queries on the same model, a hash of the model description and the performance query specification is calculated for each query, using an algorithm with a very low probability of clashes (e.g. MD5). This hashing is used to create a two-level structure in which the computed performance measures can be stored. Before any computation takes place, a cache look-up for the hash of the given model is performed. If a match is found, the hash of the current query is compared to all hashes of queries in the cache that have been evaluated on that particular model. A match indicates that the query results can be retrieved from the cache. No match means that the query needs to be evaluated. Users can configure the analysis with regards to the number of processors that are to be used during evaluation and also whether or not caching should be enabled. For more details of this mechanism, see [Harrison02, Brien08a, Brien08b].

Passage time analysis on a GSPN model is performed as described in Section 2.3.2. When PTD and Dist nodes are to be evaluated, *SMARTA* is invoked to calculate applicable pdfs and cdfs. For the computation of passage time densities, distributions and convolutions, an interesting issue arises when displaying graphs. Result graphs are often required to show data only within time ranges of relevance, i.e. only within the time bounds in which relevant probability fluctuations occur. An algorithm for the automatic determination of the time range of interest over which pdfs and cdfs should be plotted has been devised. The algorithm establishes at what time value t cdfs approach 1 within some ϵ bound. In the context of passage time queries, the probability of not reaching target states is nearly 0 beyond this point, and not likely to be of interest. Once t has been found, pdfs and cdfs are plotted between 0 and t.

Steady-state performance statistics for the GSPN model are derived by *DNAmaca* through generating and solving a CTMC that corresponds to the model's behaviour at the state-transition level. Steady-state probability distributions and high-level performance measures, such as throughput and mean buffer occupancy can be derived from these CTMCs, as shown in Section 2.3.2. These measures correspond to the FR and SS:P Performance Tree operators.

The calculation of raw moments of passage time densities and distributions, which is used for the evaluation of the Moment operator, is performed by *MOMA*. Percentiles of passage time densities and distributions are calculated by *PERC*, while their convolutions are determined by *CONE*. *PROBI* is used for obtaining the probability of a passage occurring in a given time period, as represented by the ProbInInterval operator.

Trivial computations, such as arithmetic and boolean operations and comparisons do not require dedicated analysis tools; therefore support for them has been integrated into the Analysis Server. At present, *PIPE2* allows the use of the ProbInStates, SS:S and StatesAtTime operators in the specification of performance queries; however, analysis functionality catering for their evaluation has yet to be integrated into the analysis environment.

When jobs have completed, threads forward their results to *PIPE2* for visualisation to the user.

To give an indication of the overall performance of the evaluation environment, Table 5.1 provides a comparison of observed *SMARTA* run times for the calculation of passage time densities on two models of differing sizes, as originally presented in [Dingle04a]. These run times are appropriate indicators, as passage time density calculations are generally the most time-consuming of query evaluation operations, and hence contribute most significantly to overall evaluation time. Calculations may be carried out by a varying number of processors, and Table 5.2 shows the observed gain in evaluation speed, based on the actual number of processors used. It is apparent that the most significant improvement in terms of evaluation performance can be achieved with models that have a large number of states.

Model Name	No. of States	Run times					
		1 Proc.	2 Proc.	4 Proc.	8 Proc.	16 Proc.	32 Proc.
Courier	29 010	542.7	293.6	170.6	145.6	166.6	232.8
FMS	2 519 580	27 593.8	13 790.2	6 961.3	3 548.9	1 933.4	1 079.7

Table 5.1: Run times in seconds for the evaluation of passage time densities using *SMARTA*

Model Name	Gain in evaluation speed					
	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.	32 proc.
Courier	0%	45.9%	68.6%	73.2%	69.3%	57.1%
FMS	0%	50%	74.8%	87.1%	93%	96.1%

Table 5.2: Percentage-wise gains in evaluation speed when compared to query evaluation with a sigle processor

Chapter 6

Case Studies

This chapter explores the applicability of Performance Trees by presenting a number of performance evaluation case studies. We specify Performance Tree queries on models of an electronic voting system, an online transaction system and a hospital's Accident & Emergency unit. We evaluate them and discuss obtained results.

6.1 Electronic Voting System

Below, we present a model of an electronic voting system with breakdowns and repair [Bradley03c]. The voting system is modelled by a GSPN, as shown in Figure 6.1.

In the model, voters are processed by polling units, and votes are processed by voting servers. Voters can only vote when a polling unit is available, and a vote can only be counted if a voting server is ready. Once a vote has been processed, the polling unit that has dealt with the voter casting the vote becomes available, as does the voting server that has processed the vote. Polling units and voting servers can suffer breakdowns, but can also be repaired.

We will now specify a number of performance queries to obtain insight into various performance aspects of the system, with parameters of the model being 100 voters (CC =

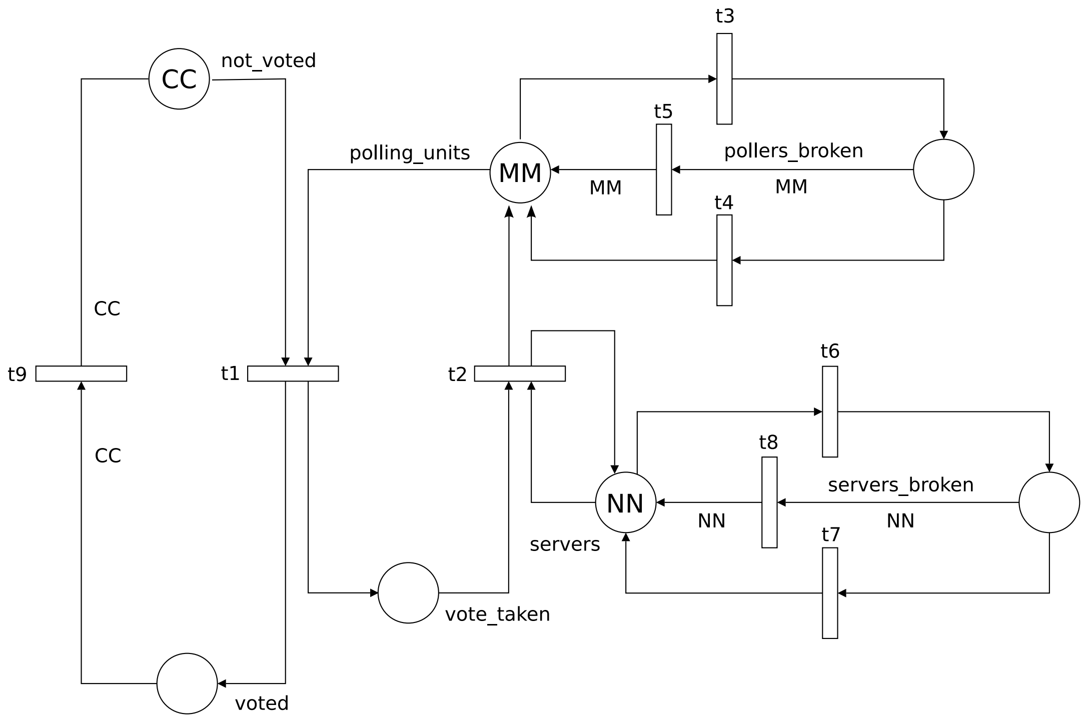


Figure 6.1: GSPN model of an Electronic Voting System

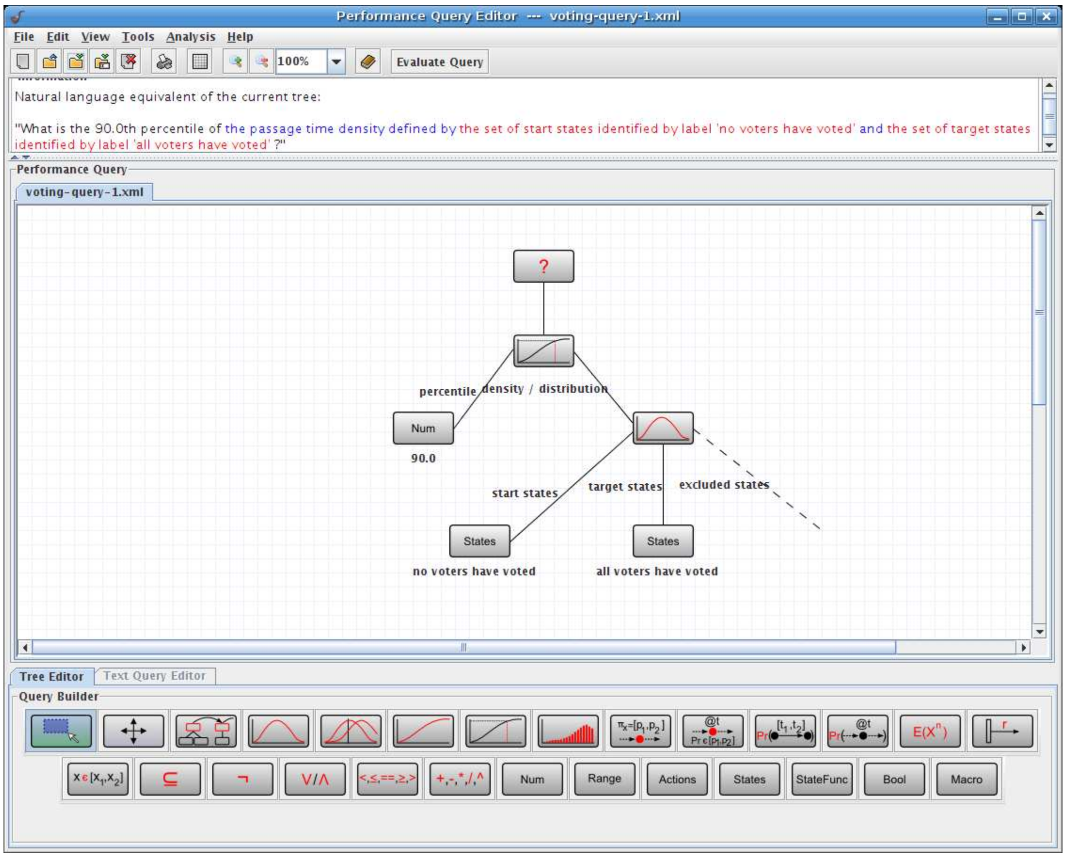
100), 10 polling units (MM = 10) and 10 servers (NN = 10). GSPN transition rates are specified in the *DNAmaca* model description in Section A.1.1.

Query 1

With this query, we attempt to gain an appreciation of the speed with which the voting system processes voters. To this end, we are interested in knowing how many minutes it takes for the system to process all voters with 90% probability, provided that nobody has voted yet at time 0.

The Performance Tree that corresponds to this query is shown in Figure 6.2. We represent the evolution of the system from the moment when no voter has voted yet to the moment when all voters have voted as a passage with the PTD operator, representing a passage time density. This operator requires start and target states to be supplied as arguments, defined by state labels. We use *'no voters have voted'* as the label for the set of start states and *'all voters have voted'* as the label for the set of target states. The constraints that are associated with these state labels, and through which the sets of states are identified, are as follows:

$$
\begin{aligned}
\textit{`no voters have voted'} &:= (\#(\text{not\_voted}) == 100) \wedge (\#(\text{polling\_units}) == 10) \wedge \\
&\quad\ (\#(\text{servers}) == 10) \\
\textit{`all voters have voted'} &:= (\#(\text{voted}) == 100)
\end{aligned}
$$


Natural language equivalent of the current tree:

"What is the 90.0th percentile of the passage time density defined by the set of start states identified by label 'no voters have voted' and the set of target states identified by label 'all voters have voted' ?"

Figure 6.2: A query addressing the 90th percentile of a passage from the state where no voters have voted to the state where all voters have voted

This query can at present not be expressed by any other query specification formalism. This is due to the fact that the sought result is a percentile of a passage time density, which is a quantitative measure that only Performance Trees are able to express.

Evaluation of the query results in the generation of a state space of 218 526 states and 1 132 483 transitions. Once evaluation has completed, we obtain the probability density function of the passage between the start and target states, as shown in Figure 6.3. During the evaluation of the Perctl operator, this density is used to calculate the 90th percentile, which is found to be 17.5 minutes. That is, 90% of the time, it takes 17.5 minutes until all voters have voted.

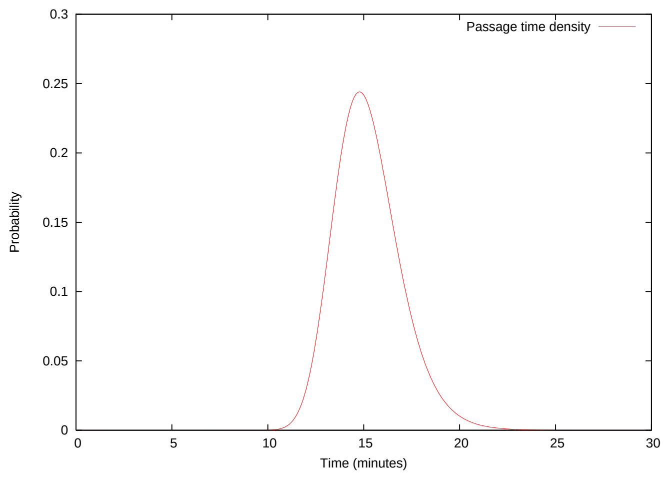


Figure 6.3: Probability density of the time taken for all voters to have voted

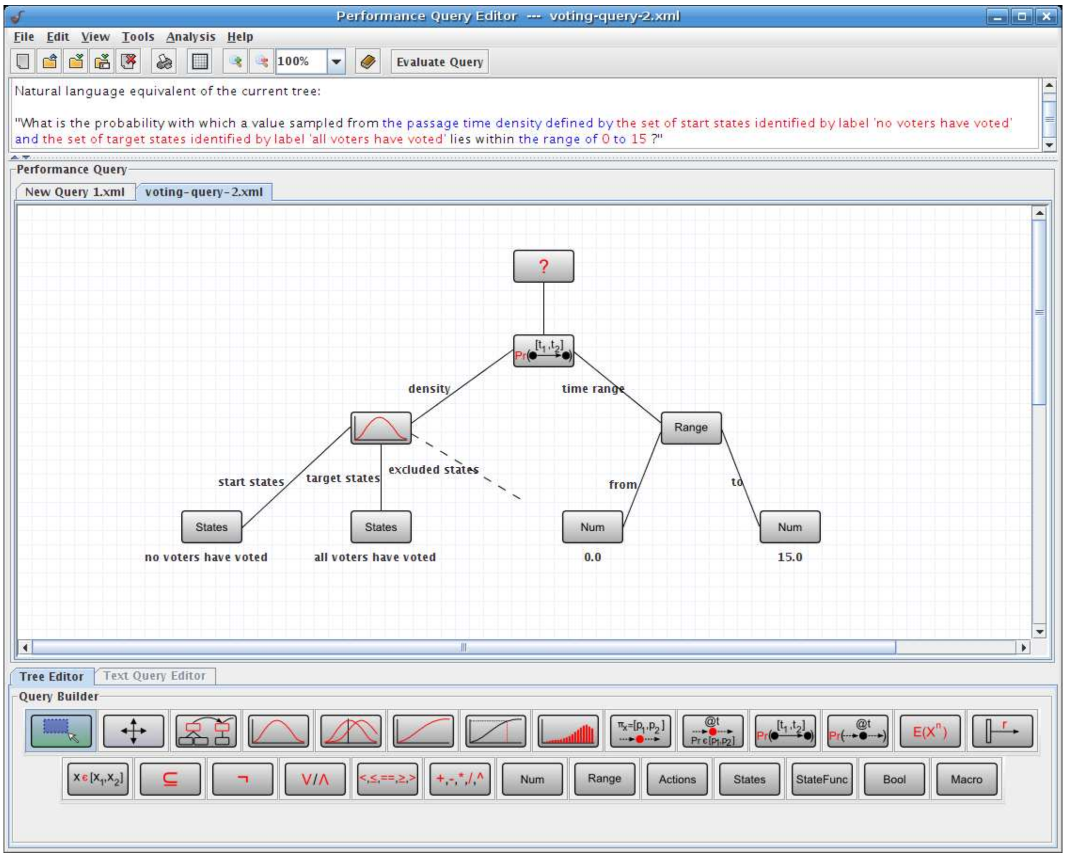
Query 2

Having already obtained an indication of the performance of the voting system, we would like to further assess its efficiency by specifying a performance query that is interested in obtaining the probability with which all voters have successfully voted within 15 minutes of the opening of the polling stations.

In Performance Tree form, this query is represented as shown in Figure 6.4. In this query, we use the same passage as in the previous query to represent the system transitioning from the initial state to the state in which all voters have voted.

This query cannot be expressed in other specification formalisms, due to the need to reason about a passage time density, which at present only Performance Trees are able to.

During evaluation, the passage time density for all voters to have voted is calculated, which is the same as for the previous query (shown in Figure 6.3). If this query were to be evaluated after the previous query has already been evaluated, the Analysis Server would realise that the passage time density that is to be calculated is identical to the density that has already been calculated for the previous query, and hence results would be retrieved from the cache. During the evaluation of the ProbInInterval operator, the probability with which all voters have voted within 15 minutes is calculated from the passage time density. This probability is found to be 0.483. Given that 15 minutes is a very short period of time to process 100 voters, the fact that all voters can be processed by the system almost 50%



Natural language equivalent of the current tree:

"What is the probability with which a value sampled from the passage time density defined by the set of start states identified by label 'no voters have voted' and the set of target states identified by label 'all voters have voted' lies within the range of 0 to 15 ?"

Figure 6.4: A query addressing the probability with which all voters have voted within 15 minutes

of the time indicates that it is rather efficient.

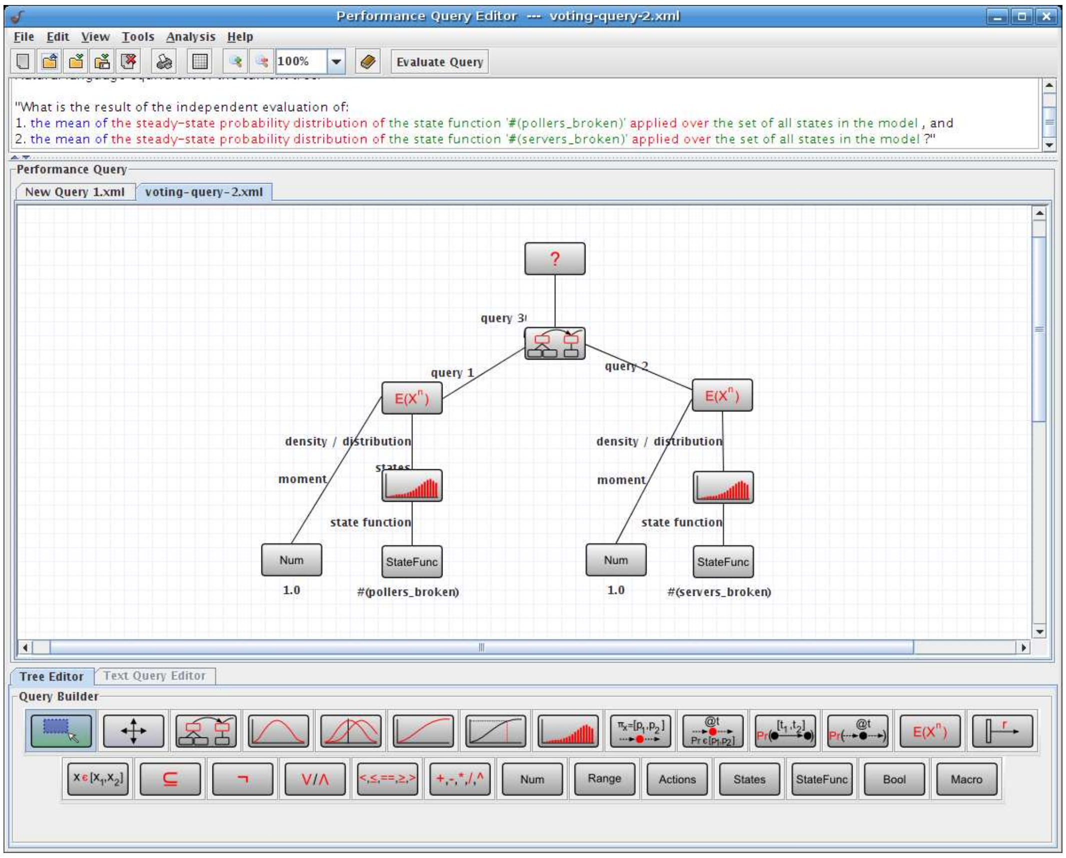
Query 3

To assess the reliability of the voting system, we are interested in the average number of broken polling units and broken servers on the long run.

We specify this as a Performance Tree query as shown in Figure 6.5. In this query, we make use of the Mult operator, which allows us to specify compound performance queries.

This query cannot be expressed in other specification formalisms, due to the need to compute the expectation of a steady-state probability distribution, which at present is not supported by any other formalism.

This is the first performance query in our case study so far that can be evaluated in parallel. The Mult operator combines two independent queries into one, which implies that they



"What is the result of the independent evaluation of:
1. the mean of the steady-state probability distribution of the state function '#(pollers_broken)' applied over the set of all states in the model , and
2. the mean of the steady-state probability distribution of the state function '#(servers_broken)' applied over the set of all states in the model ?"

Figure 6.5: A query addressing the average number of broken polling units and servers at steady-state

can safely be evaluated in parallel, since there are no dependencies between them. During evaluation of the query, we obtain the steady-state probability distribution for the number of broken polling units, as shown in Figure 6.6. From this distribution, we also obtain the average number of broken polling units, which is found to be 0.567. Evaluation also produces the steady-state probability distribution for the number of broken servers, which is shown in Figure 6.7. The average number of broken servers is found to be 0.196. Considering these results, we can confidently claim that the voting system is reliable, since on average, a low number of polling units and servers fail.

Query 4

To obtain another indication of system reliability, we formulate a performance query that aims to obtain the expected time until two voting servers have broken down.

The Performance Tree equivalent of this query is shown in Figure 6.8; relevant state labels

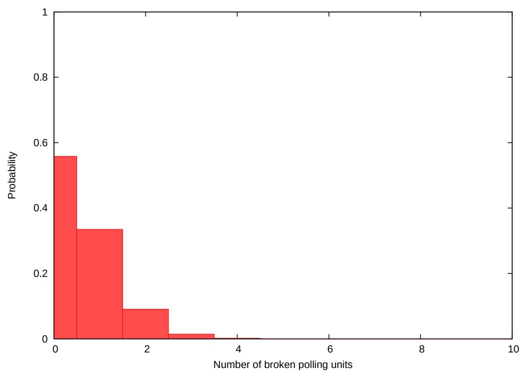


Figure 6.6: Steady-state distribution of the number of broken polling units in the voting system

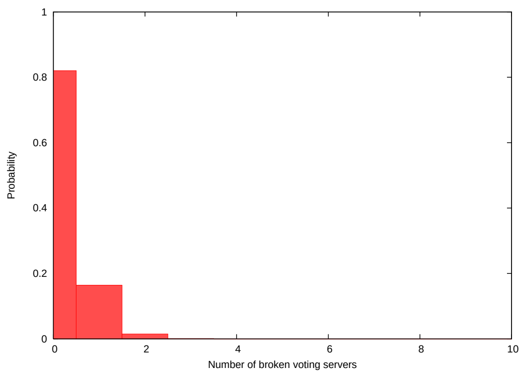


Figure 6.7: Steady-state distribution of the number of broken servers in the voting system

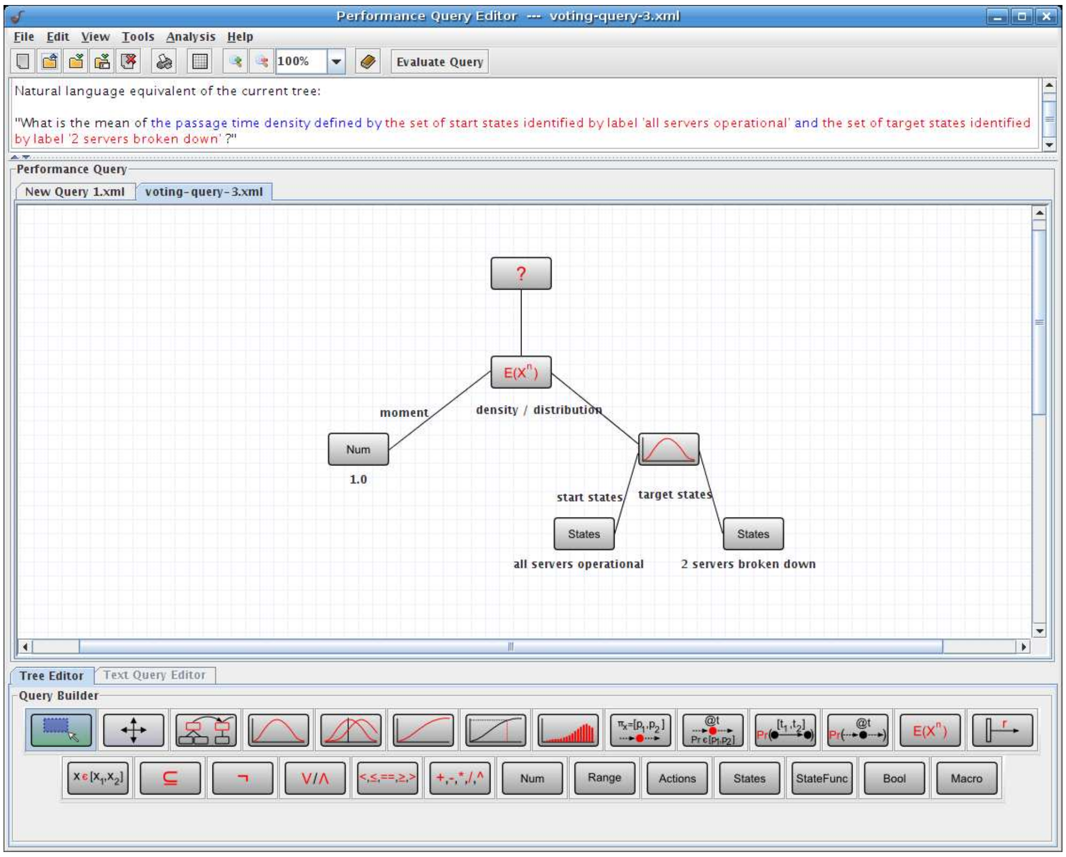
are defined as:

$$\begin{aligned} \textit{`all servers operational'} &:= (\#(\text{not\_voted}) == 100) \wedge (\#(\text{servers}) == 10) \wedge \\ & \quad\; (\#(\text{polling\_units}) == 10) \\ \textit{`2 servers broken down'} &:= (\#(\text{server\_broken}) == 2) \end{aligned}$$

In the performance query, we represent with a passage time density the moving of the system from the state where all voting servers are operational to a state where two voting servers are broken. This is shown in Figure 6.9. This function represents the probability density of the time it takes for two servers to break down, given that all servers have been operational at the time when observation has commenced. From the density we can obtain the expected time until two voting servers break down. This value is found to be 43.33 minutes, which indicates that the system is fairly robust.

6.2 Online Transaction System

In this case study, we revisit the Online Transaction System of Section 3.3, the GSPN model of which is shown in Figure 3.9. We will evaluate some of the queries presented earlier, along with several new queries. We parameterise the model with eight customers, all of whom are initially assumed to be browsing randomly on the Internet. Transition rates for the GSPN are specified in the *DNAmaca* model description in Section A.2.1.


Natural language equivalent of the current tree:

"What is the mean of the passage time density defined by the set of start states identified by label 'all servers operational' and the set of target states identified by label '2 servers broken down' ?"

Figure 6.8: A query addressing the expected time until two voting servers have broken down

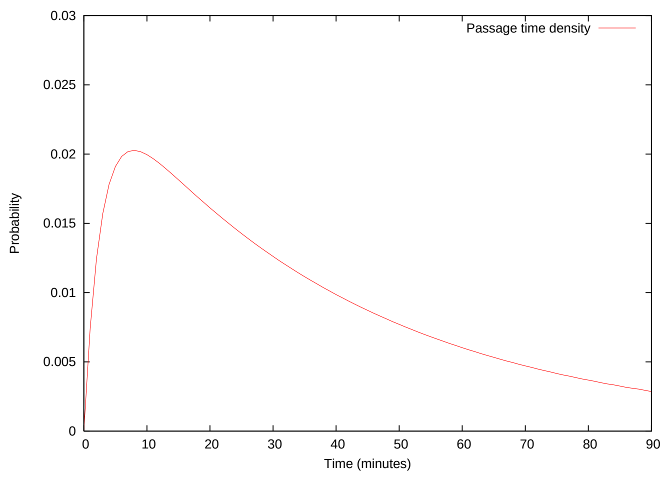


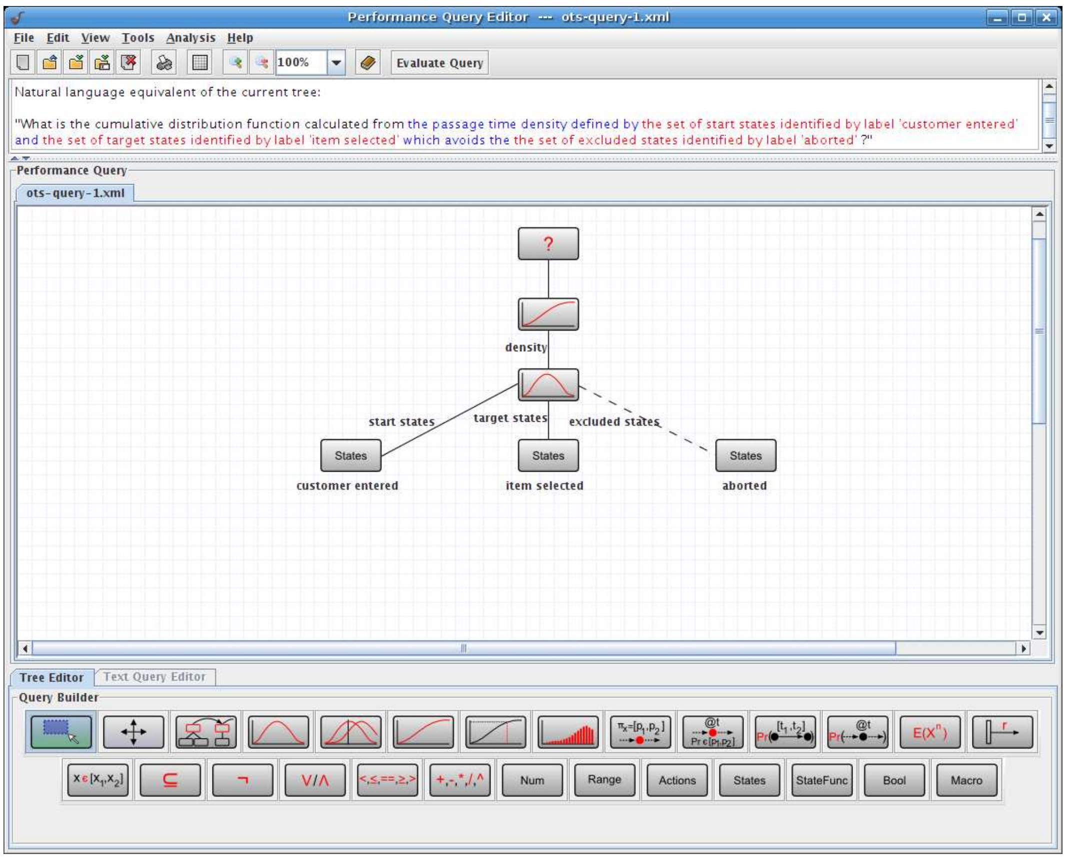
Figure 6.9: The probability density of the time needed for two voting servers to break down

Query 1

In our first query, we are interested in the distribution of time for a customer to select an item from the catalogue, starting from the moment when they have entered the web site, and assuming that they have not left in the meantime. This can be useful for assessing the design quality of the web site, from the point of view of how easily customers are able to navigate to the product catalogue.

The Performance Tree corresponding to the query is shown in Figure 6.10, and applicable state labels are defined as:

'customer entered'	:=	(tag@(site entered))
'item selected'	:=	(tag@(item selected))
'aborted'	:=	(tag@(transaction aborted))


Natural language equivalent of the current tree:

"What is the cumulative distribution function calculated from the passage time density defined by the set of start states identified by label 'customer entered' and the set of target states identified by label 'item selected' which avoids the the set of excluded states identified by label 'aborted' ?"

Figure 6.10: A query addressing the distribution of time taken for a customer to select an item from the product catalogue after having entered the web site

This query cannot be expressed by other specification formalisms at present, due to the need to reason about a probability density and distribution, as well as individual customers in the system.

Evaluation of the query yields a state space of 213 928 states and 2 580 864 transitions. To obtain the distribution that we are interested in (see Figure 6.12), the passage time density needs to be calculated first. This is shown in Figure 6.11.

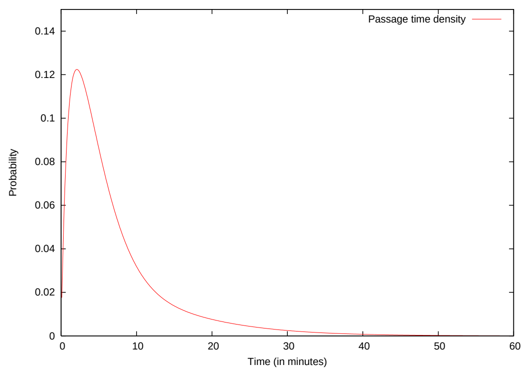


Figure 6.11: Probability density of the time taken for a customer to select an item from the catalogue

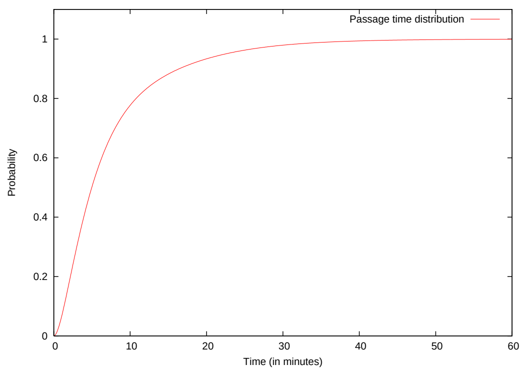


Figure 6.12: Probability distribution of the time taken for a customer to select an item from the catalogue

Query 2

With this query, we are aiming to assess the speed with which customers make purchases at the web site by obtaining the probability of an order being placed within 10 minutes of a customer having entered the web site, provided that they have not left and returned in the meantime.

The Performance Tree equivalent of this query is shown in Figure 6.13, and relevant state labels are defined as:

‘*customer entered*’	:=	(tag@(site entered))
‘*order confirmed*’	:=	(tag@(order confirmed))
‘*aborted*’	:=	(tag@(transaction aborted))

This query can also not be expressed using other formalisms, due to the need to obtain the probability with which a passage occurs in a given amount of time, and the necessity to reason about a single customer’s flow through the system. These features are only supported by Performance Trees at present.

Evaluation obtains the passage time density of an order being placed by a customer after

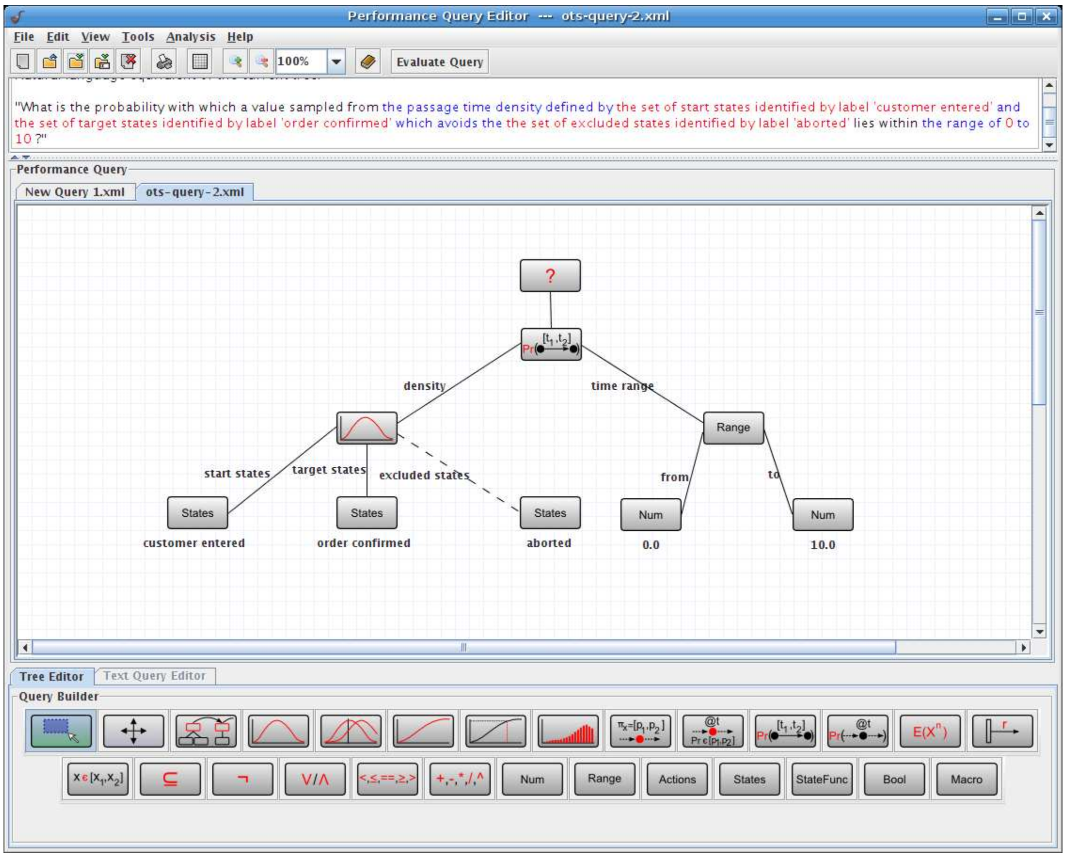

"What is the probability with which a value sampled from the passage time density defined by the set of start states identified by label 'customer entered' and the set of target states identified by label 'order confirmed' which avoids the the set of excluded states identified by label 'aborted' lies within the range of 0 to 10 ?"

Figure 6.13: A query addressing the probability with which an order has been confirmed within 10 minutes of a customer having entered the web site

having entered the web site. This is shown in Figure 6.14. From this density, we find that the probability of the order being placed by a customer within 10 minutes is 0.176. The fact that only 17.6% of visitors complete an order within 10 minutes may indicate that either most visitors only browse the web site without purchasing anything or that customers generally take a while to browse the catalogue before committing themselves to a purchase.

Query 3

With this query, we want to find out whether in 90% of the cases, the time that it takes a single customer to enter the web site and proceed to the checkout, and to provide their billing information for the order to complete and either leave the site or return to the catalogue is less than 15 minutes, provided that they have not aborted the transaction in the meantime.

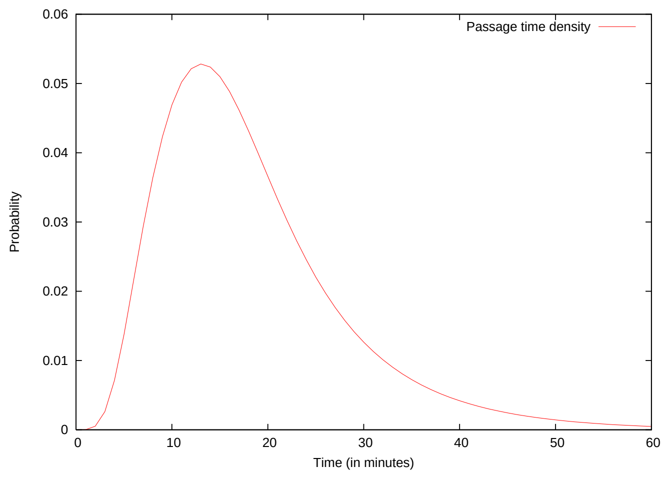


Figure 6.14: Passage time density for an order having been placed after a customer has entered the site

The Performance Tree equivalent of this query is shown in Figure 6.15, and state labels used in the query are defined as:

'start1'	:=	(tag@(site entered))
'target1'	:=	(tag@(at checkout))
'start2'	:=	(tag@(billing info provided))
'target2'	:=	(tag@(not at site)) $\vee$ (tag@(browsing catalogue))
'aborted'	:=	(tag@(transaction aborted)))

This query cannot be reproduced by any other specification formalism, since it requires the ability to reason about passage time densities, their percentiles and convolutions, which are features specific to Performance Trees at the moment.

When the aim is to focus only on parts of a particular passage and disregard a period of time in between, convolutions are useful. In this query, we are only interested in how long it takes the customer to reach the checkout from the moment when they have entered the web site and how long it takes them to either leave or return to the catalogue after they have provided their billing information. We are not interested in the amount of time that elapses between them reaching checkout and providing their billing information.

Hence, during evaluation we calculate the passage time density for a customer to arrive at the checkout, starting from their arrival at the site, as shown in Figure 6.16. We also calculate the passage time density for the same customer to have left the web site or

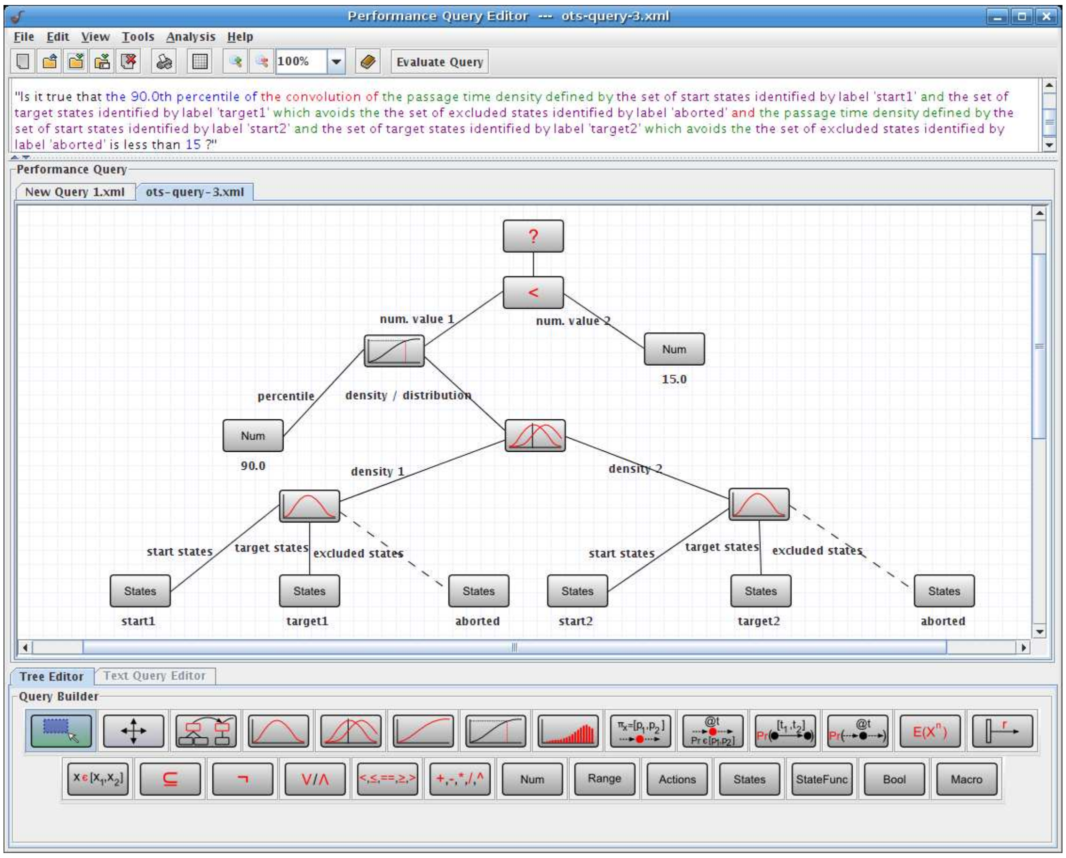

"Is it true that the 90.0th percentile of the convolution of the passage time density defined by the set of start states identified by label 'start1' and the set of target states identified by label 'target1' which avoids the the set of excluded states identified by label 'aborted' and the passage time density defined by the set of start states identified by label 'start2' and the set of target states identified by label 'target2' which avoids the the set of excluded states identified by label 'aborted' is less than 15 ?"

Figure 6.15: A query addressing a constraint on the 90th percentile of the convolution of two passage time densities for a customer to enter the site and proceed to the checkout and to provide their billing information and leave the site or return to the product catalogue

returned to the product catalogue from the moment when they have provided their billing information, as shown in Figure 6.17. These calculations can be carried out in parallel, since they are independent. We convolve the two densities to represent the combined passage time density of the two independent passages. This is shown in Figure 6.18. We then calculate the 90th percentile of this convolved density, which we find to be 22.16. This indicates that 90% of the time, it takes a customer more than 15 minutes to enter the site and proceed to the checkout, and to provide their billing information and leave the site or return to the product catalogue. Therefore, the result of the query is *false*.

Query 4

With this query, we aim at assessing the popularity of the web site by looking at the average rate at which visitors enter the site. We also want to find out how many of the visitors are not only looking around, but are in fact searching for a product in the catalogue.

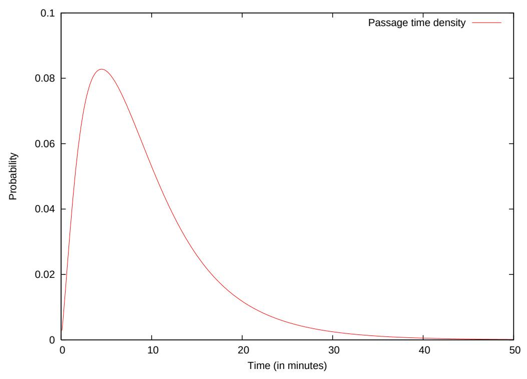


Figure 6.16: Passage time density for a customer to have arrived at the checkout, starting from the moment of their arrival at the web site

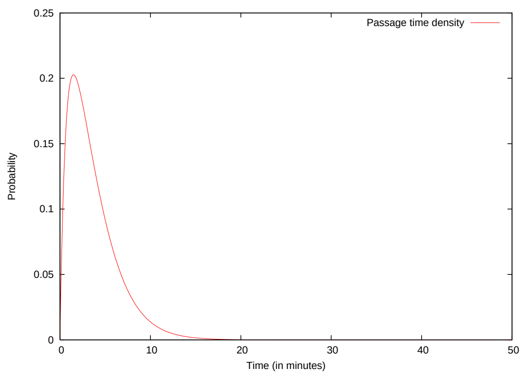


Figure 6.17: Passage time density for a customer to have left the web site, starting from the moment when they have provided their billing information

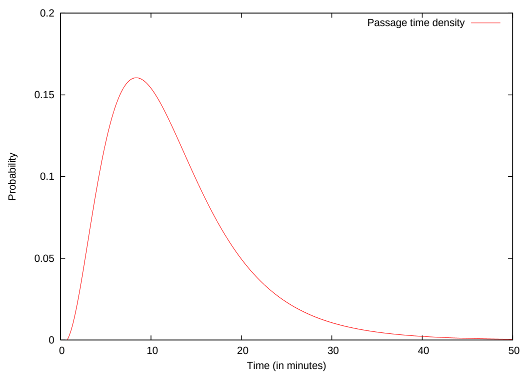


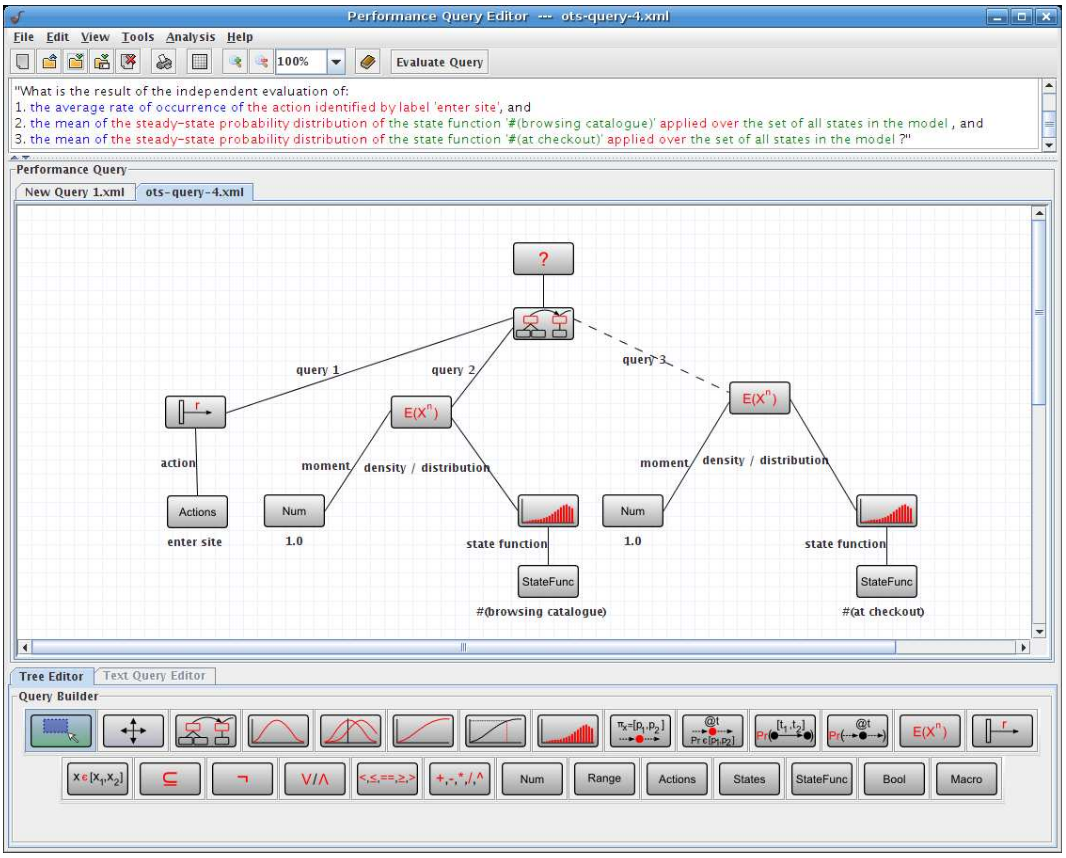
Figure 6.18: Convolution of the passage time densities of Figure 6.16 and Figure 6.17

Hence we are also interested in the average number of customers browsing the catalogue. In addition, to find out how many customers make purchases after browsing the catalogue on average, we are also interested in obtaining the average number of customers at the checkout. The Performance Tree that represents this query is shown in Figure 6.19.

No other specification formalisms are currently able to express this query, due to the need to reason about the average throughput of a transition and the expected values of steady-state probability distributions.

As the query consists of three independent sub-queries that are linked together by the Mult operator, it can be evaluated in parallel. Its evaluation finds the average number of visitors entering the web site to be 0.404 persons per minute.

Considering the size of our model's population (there are eight people in the system at



"What is the result of the independent evaluation of:
1. the average rate of occurrence of the action identified by label 'enter site', and
2. the mean of the steady-state probability distribution of the state function '#(browsing catalogue)' applied over the set of all states in the model , and
3. the mean of the steady-state probability distribution of the state function '#(at checkout)' applied over the set of all states in the model ?"

Figure 6.19: A query addressing the average rate of occurrence of customers entering the web site, the average number of customers browsing the catalogue, and the average number of customers at the checkout

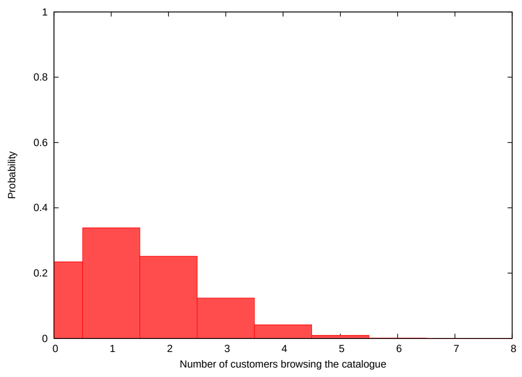


Figure 6.20: Steady-state distribution of the number of customers browsing the product catalogue

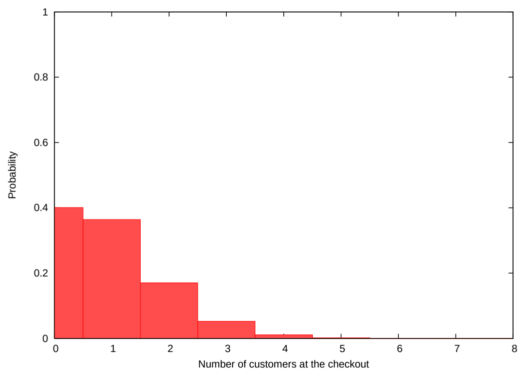


Figure 6.21: Steady-state distribution of the number of customers at the checkout

any given time), this appears in relative proportion to be a reasonable approximation of the traffic that a web site on the Internet might experience. The steady-state distribution of the number of visitors browsing the catalogue is shown in Figure 6.20, from which the average is calculated to be 1.432 persons per minute. Similarly, Figure 6.21 shows the steady-state distribution of customers at the checkout, which produces an average of 0.913 persons per minute. This shows that the majority of visitors who are browsing the catalogue also end up purchasing something eventually. Thus, revisiting our interpretation of the results for Query 2, we can now see that most visitors do purchase goods; however, it takes them more than 10 minutes to browse the product catalogue and place an order.

6.3 Hospital Accident & Emergency Unit

In this case study, we will construct and analyse performance queries on a modified version of the hospital Accident & Emergency (A&E) unit model first introduced in [Suto07].

The GSPN model of the A&E unit is shown in Figure 6.22. The model describes a system with the following behaviour. An initial number of healthy individuals fall ill at a certain rate, and it is pessimistically assumed that they will need to visit an A&E unit at a local hospital. Individuals who have fallen ill either go to the hospital themselves, in which case they are categorised as walk-in patients, or place an emergency call in acute cases to request an ambulance. In this case, they are classified as ambulance patients. After having entered A&E, walk-in patients are asked to wait until they can be seen by a nurse for initial assessment. Ambulance patients are loaded onto a trolley, on which they wait until a nurse becomes available to attend to them. Nurses assess ambulance patients with priority. After initial assessment, patients proceed to wait to either be seen by a doctor, be taken for emergency surgery, or be sent for laboratory tests. Once a patient is discharged, they are optimistically assumed to be healthy.

The model is parameterised with P, N, D and A, which denote the number of tokens on places *healthy*, *nurses*, *doctors* and *ambulances*, respectively. For our case study scenario, we use the configuration $P = 8$, $N = 2$, $D = 2$ and $A = 1$. GSPN transition rates are specified in the *DNAmaca* model description in Section A.3.1.

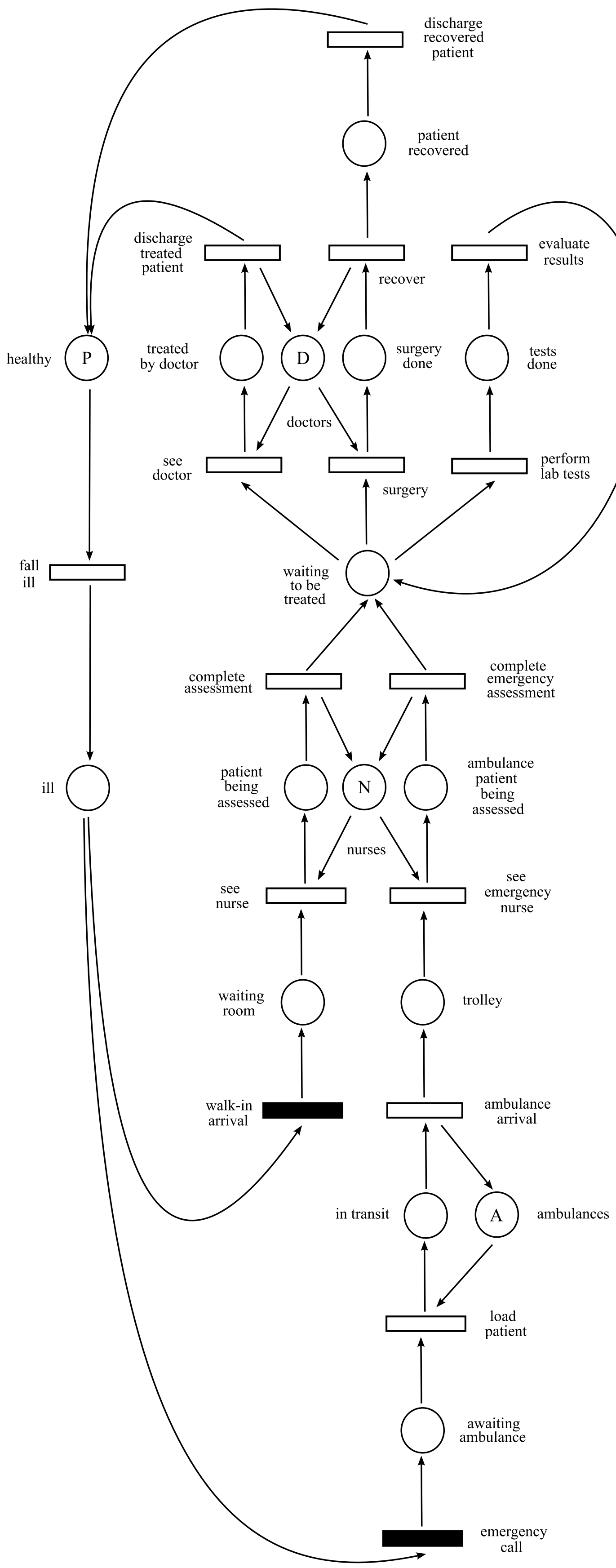


Figure 6.22: GSPN model of a hospital’s Accident & Emergency unit

Query 1

With this query, we aim to compare treatment times between walk-in and ambulance patients. Ambulance patients receive prioritised attention during the first stages of their stay at the A&E unit, as indicated by higher rates for certain transitions in the model that deal with ambulance patients, e.g. *emergency call*, *load patient*, *ambulance arrival*, *see emergency nurse* and *complete emergency assessment* (see Section A.3.1). We would like to ascertain whether this initial prioritisation has a prolonged effect on patient treatment times. If it does, most ambulance patients would be discharged from the hospital more promptly than regular walk-in patients. Hence, we specify a performance query to address the 90th percentile of the density of time taken for a walk-in patient to leave the hospital, considering from the moment of arrival. The query also addresses the same percentile of the density of time taken for an ambulance patient to leave the hospital, considering from the moment when they have called for an ambulance. If it is the case that the time value that is the 90th percentile for walk-in patients is greater than that for ambulance patients, we know that ambulance patients receive prioritised care throughout their stay at the A&E unit. In addition, to assess the efficiency of the A&E unit, we are also interested in the steady-state probability distribution of idle nurses and doctors.

The Performance Tree equivalent of this query is given in Figure 6.23. To obtain the passage time density for a single patient to proceed through A&E, we need to tag an individual customer in our model to track their progress. Relevant state labels for the model are defined as:

‘*patient in waiting room*’	:=	(tag@(waiting room))
‘*patient awaiting ambulance*’	:=	(tag@(awaiting ambulance))
‘*patient healthy*’	:=	(tag@(healthy))

This query cannot be expressed by any other specification formalism, since Performance Trees are currently the only formalism able to reason about the performance concepts forming part of this query.

Parallelisation is possible when evaluating the arithmetic comparison operator, since its two sub-trees are independent of each other, and also when evaluating the SS:P operators

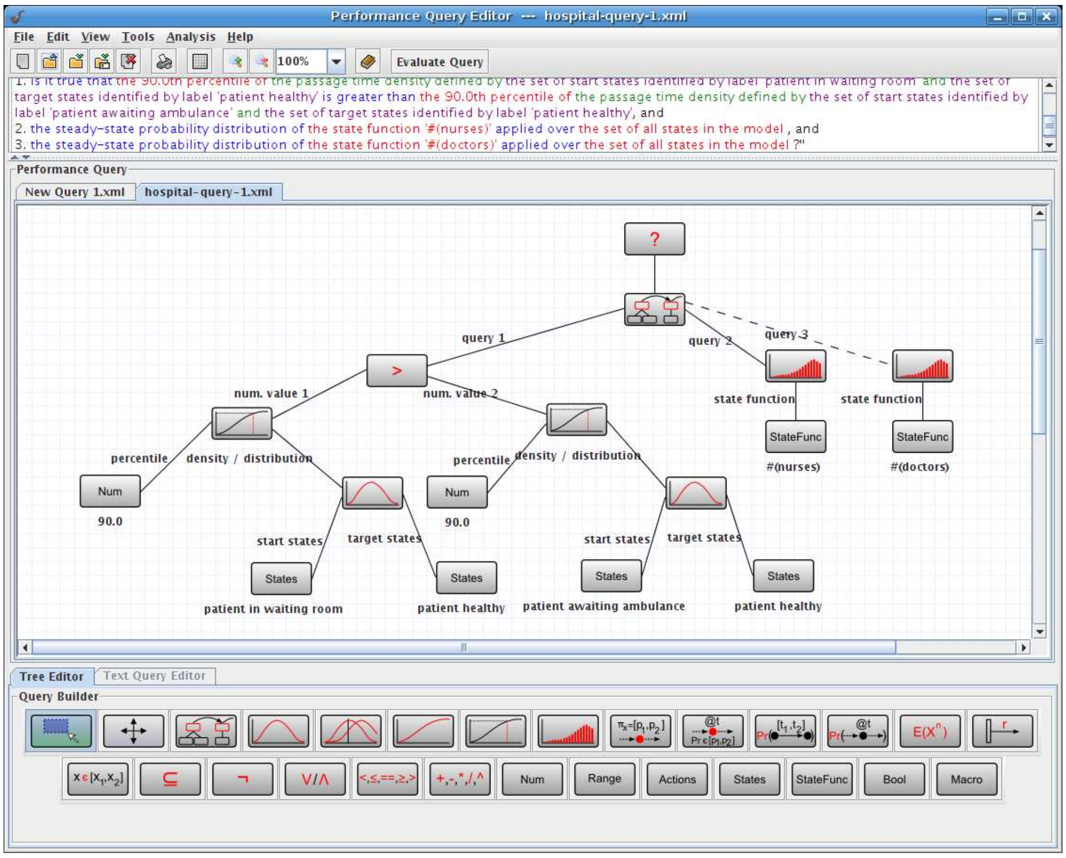

1. Is it true that the 90.0th percentile of the passage time density defined by the set of start states identified by label 'patient in waiting room' and the set of target states identified by label 'patient healthy' is greater than the 90.0th percentile of the passage time density defined by the set of start states identified by label 'patient awaiting ambulance' and the set of target states identified by label 'patient healthy', and
2. the steady-state probability distribution of the state function '#(nurses)' applied over the set of all states in the model , and
3. the steady-state probability distribution of the state function '#(doctors)' applied over the set of all states in the model ?"

Figure 6.23: Compound performance query addressing percentiles of passage time densities and steady-state probability distributions

that are connected by the Mult operator, since they are independent sub-queries.

The evaluation of this query results in the generation of a state space of 1 355 166 states and 5 483 010 transitions for the model. Evaluation determines the result of the first part of the query to be *false*, indicating that ambulance patients do not benefit from prioritised care throughout their stay at the A&E unit. This is the case because the 90th percentile of the passage time density for walk-in patients (see Figure 6.24) was found to be 72.764 minutes, while the 90th percentile of the passage time density for ambulance patients (see Figure 6.25) evaluated to 73.986 minutes. This result implies that an initial prioritisation has no overall effect on how patients are dealt with on the long run. This conclusion can also be derived intuitively by comparing the two densities and noting that there is only a marginal difference between them.

The result of the second part of the query is the steady-state probability distribution of idle nurses (see Figure 6.26), while the result of the third sub-query is the steady-state

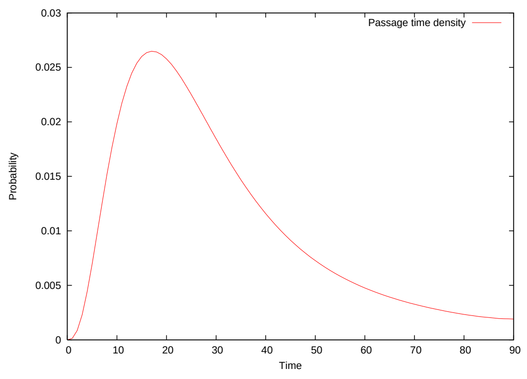


Figure 6.24: Passage time density for walk-in patients

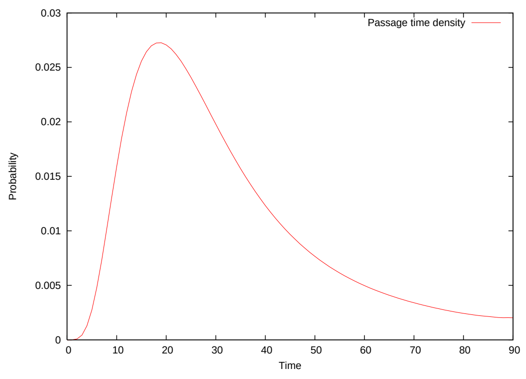


Figure 6.25: Passage time density for ambulance patients

probability distribution of idle doctors (see Figure 6.27). These results indicate that both nurses are idle more than half of the time, while doctors are kept fairly busy. Given that the number of patients is somewhat modest, this outcome is not unexpected.

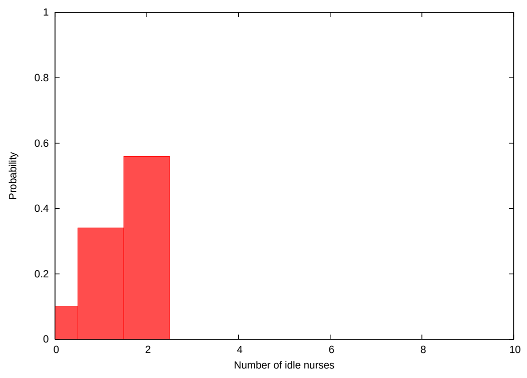


Figure 6.26: Steady-state probability distribution of idle nurses

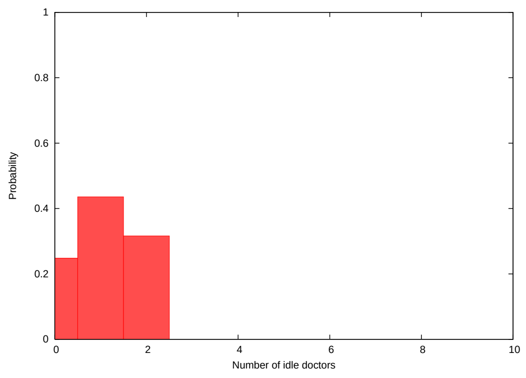


Figure 6.27: Steady-state probability distribution of idle doctors

Query 2

With this query, we would like to gain an appreciation of the efficiency of doctors at the A&E unit by assessing the average number of patients that are waiting for a doctor. In addition, we are also interested in the average rate of surgeries to assess how frequently critical care is provided at the A&E unit. The Performance Tree equivalent of this query is shown in Figure 6.28.

This query can at present only be expressed by Performance Trees, due to the need to reason about a steady-state distribution and its mean.

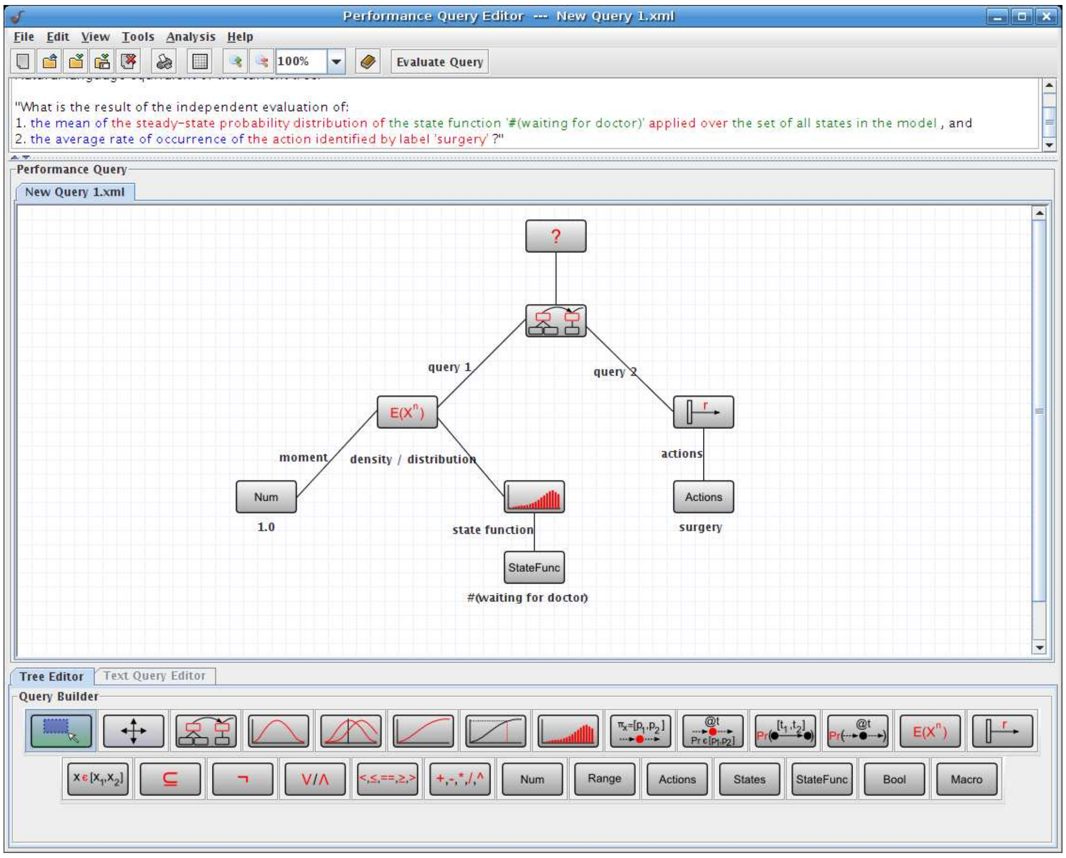

"What is the result of the independent evaluation of:
1. the mean of the steady-state probability distribution of the state function '#(waiting for doctor)' applied over the set of all states in the model , and
2. the average rate of occurrence of the action identified by label 'surgery' ?"

Figure 6.28: Performance query addressing the average number of patients waiting for a doctor and the average rate of occurrence of surgeries

The query can be evaluated in parallel, as the Mult operator connects two independent sub-queries. Hence, the evaluation of the mean of the steady-state distribution can take place at the same time as the evaluation of the average rate of surgeries.

During the evaluation of the query, a state space of 698 922 states and 5 863 182 transitions was generated. The reason for the state space being smaller than that generated previously is that for this query it is not necessary to tag customers. Figure 6.29 shows the steady-state distribution of the number of patients waiting for a doctor, whose average was calculated to be 1.25. It appears that the average number of patients waiting for a doctor is relatively low, which indicates that doctors are dealing with patients efficiently, since waiting patients are not accumulating. Results for the steady-state distribution of idle doctors that we have obtained during the evaluation of the previous query (see Figure 6.27), show that doctors are kept relatively busy. This supports our interpretation of the results of this query. From the evaluation of the second part of the query, we find that the average rate of occurrence of surgeries is 0.059 operations per hour. The fact that the average rate of surgeries is very low might be an indication for the fact that only a small proportion of hospital patients have injuries severe enough to require immediate surgery. Therefore, we conclude that critical care only needs to be provided in a very small number of cases.

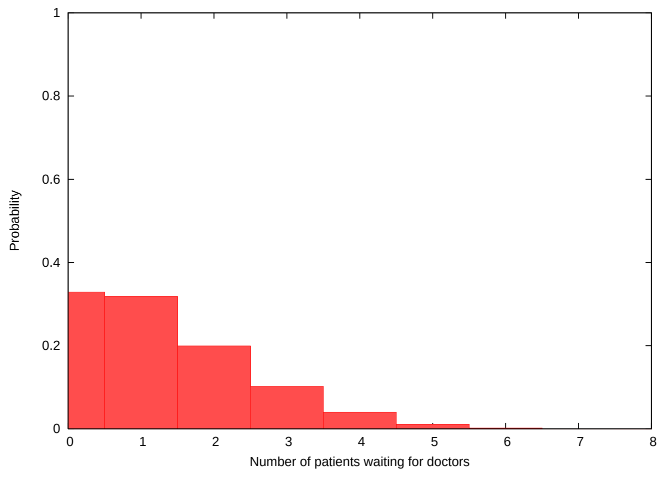


Figure 6.29: Steady-state probability distribution of the number of patients waiting for a doctor

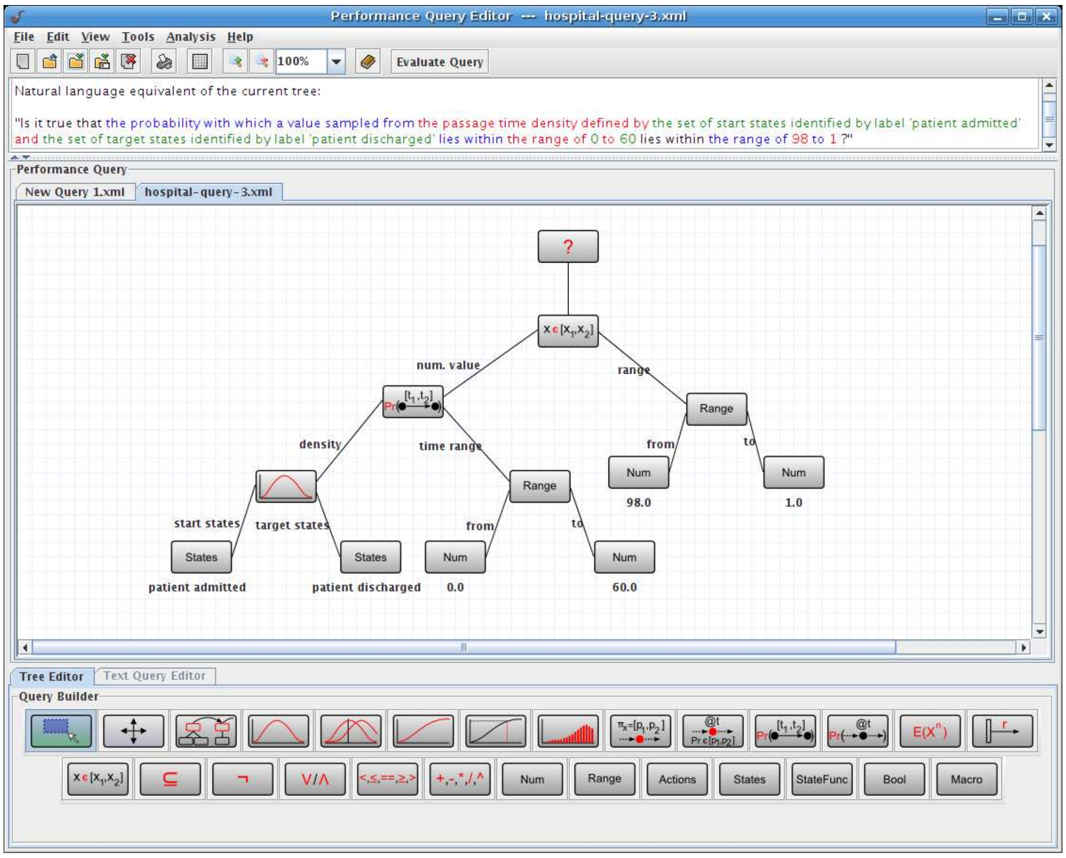
Query 3

In this query, we establish a requirement for the A&E unit, stating that 98% of the time, all patients should be seen, treated and discharged within an hour. The Performance Tree equivalent of this query is shown in Figure 6.30, with applicable state labels defined as:

'patient admitted'	$:=$	(tag@(waiting room)) $\vee$ (tag@(trolley))
'patient discharged'	$:=$	(tag@(healthy))

Other query specification formalisms are not able to express this query, as it requires reasoning about a passage time density and the probability with which a passage occurs within a given time period.

Due to the requirement of this query to reason about tagged customers, state space generation results in the same number of states and transitions as for the first query. Evaluation of the query yields a probability density function for the time taken for patients to enter A&E, be seen, treated and discharged. This is shown in Figure 6.31. The probability with which this passage takes place within one hour (i.e. 60 minutes) is calculated to be 0.863,



Natural language equivalent of the current tree:

"Is it true that the probability with which a value sampled from the passage time density defined by the set of start states identified by label 'patient admitted' and the set of target states identified by label 'patient discharged' lies within the range of 0 to 60 lies within the range of 98 to 1 ?"

Figure 6.30: Performance query addressing a modified version of the UK Government target for A&E units

which does not lie within the probability interval $[0.98, 1]$, as set out by the query. Hence, the result of the query is *false*, since patients are not seen, treated and discharged within an hour 98% of the time.

Query 4

With this query, we aim at obtaining the Coefficient of Variation (the ratio of the standard deviation to the mean) of the time that it takes for the first person to recover from surgery at the A&E unit.

This query can be specified using the Performance Tree macro mechanism. The macro definition for the concept of Coefficient of Variation is shown in Figure 6.32, and its usage with the argument applicable to this query is presented in Figure 6.33. Relevant state labels are defined as:

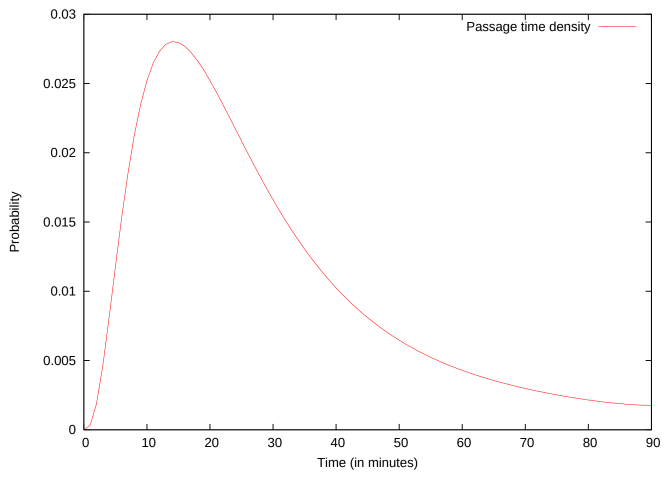
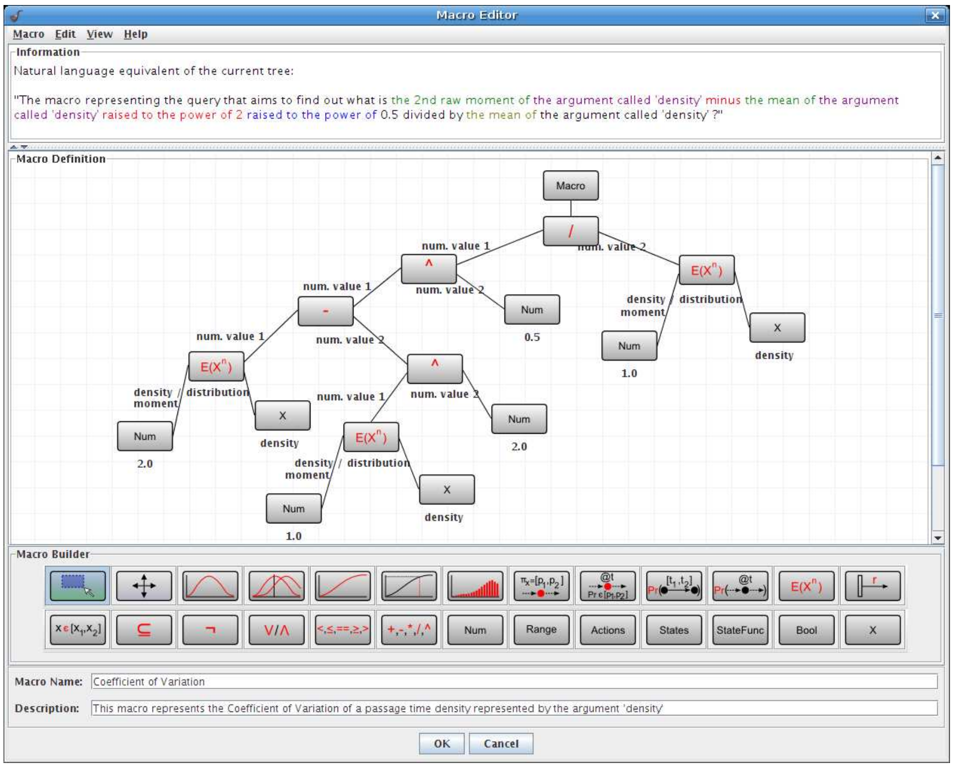


Figure 6.31: Density of the time taken for patients to enter and leave the A&E unit

'everyone healthy'	:=	(#(healthy) == 8) ∧ (#(nurses) == 2) ∧
		(#(doctors) == 2) ∧ (#(ambulances) == 1)
'first patient recovered'	:=	(#(patient recovered) == 1)



Natural language equivalent of the current tree:

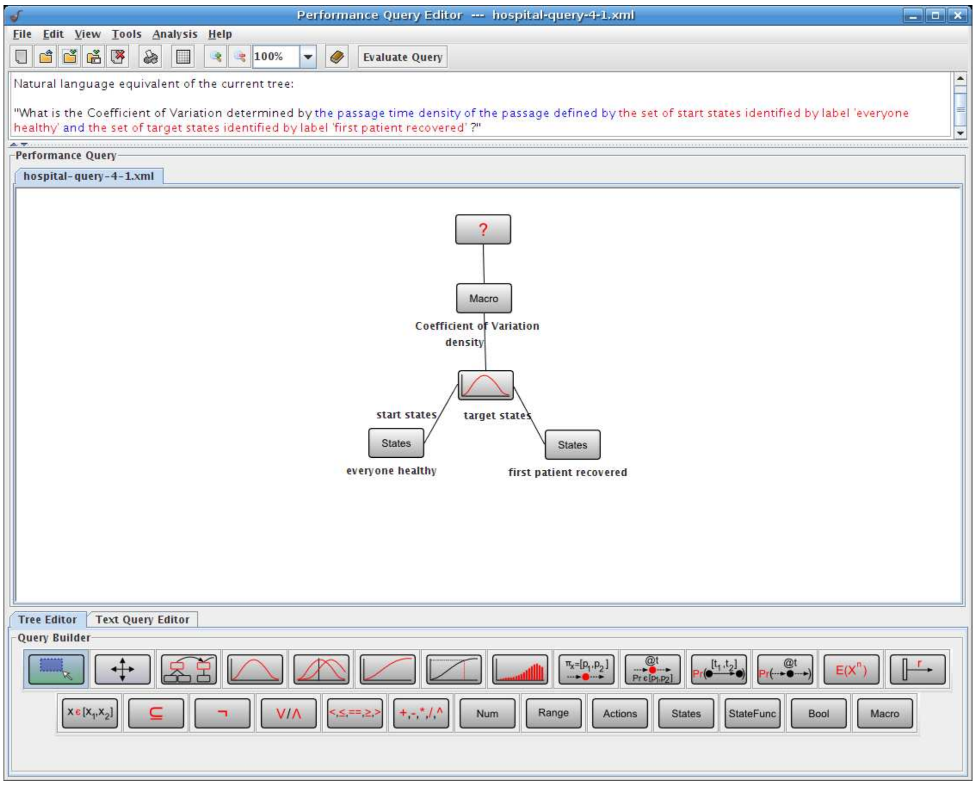
"The macro representing the query that aims to find out what is the 2nd raw moment of the argument called 'density' minus the mean of the argument called 'density' raised to the power of 2 raised to the power of 0.5 divided by the mean of the argument called 'density' ?"

Macro Name: Coefficient of Variation

Description: This macro represents the Coefficient of Variation of a passage time density represented by the argument 'density'

Figure 6.32: Definition of the macro representing the concept of Coefficient of Variation

This query can only be expressed by Performance Trees, due to the need to reason about concepts that only they are able to express.


Natural language equivalent of the current tree:

"What is the Coefficient of Variation determined by the passage time density of the passage defined by the set of start states identified by label 'everyone healthy' and the set of target states identified by label 'first patient recovered' ?"

Figure 6.33: Usage of the macro for the calculation of the Coefficient of Variation with a specified argument

This query is a good example of how caching can be used to increase evaluation speed, since the mean of the passage time density is used twice. Once it has been evaluated, the result can be retrieved from the cache when it is be calculated the second time. During the evaluation of the query, the same number of states and transitions were generated as for the first query. The density of time for the first patient to have recovered is shown in Figure 6.34. Its mean and variance are found to be 27.4982 minutes and 1005.51 minutes^2, respectively. From these values, we obtain the Coefficient of Variation as being 0.5743.

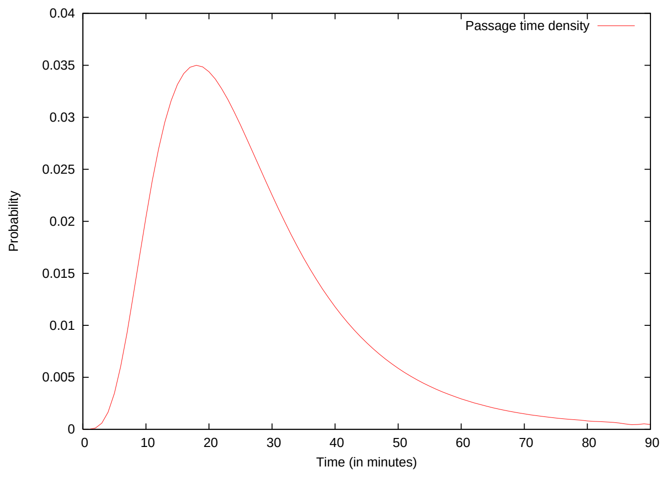


Figure 6.34: Passage time density for the first a patient to recover at the A&E unit

Chapter 7

Conclusion

7.1 Conceptual Contributions

The main contribution of the work presented in this thesis is the introduction and development of the Performance Tree formalism – a new approach to the specification of performance queries on models of real-life systems.

Until recently, performance queries have mostly been specified with complicated textual languages that require specialist knowledge to be used correctly. Performance Trees offer an accessible graphical alternative that assumes only a basic engineering background and that enables users to specify performance queries intuitively, without the need to learn complicated syntax or sophisticated abstract concepts. Performance queries are specified as hierarchical tree structures, consisting of a set of nodes that represent performance concepts and values, and a set of arcs that connect them. The hierarchical node structure resembles that of a computer program in the way in which nodes representing performance concepts behave like functions that take inputs and produce an output, and the way in which nodes representing values are interpreted as inputs to such functions. This hierarchical structure makes the specification of Performance Tree queries rather intuitive, and also allows queries to become arbitrarily complex, while at the same time maintaining a reasonable level of clarity and manageability from a user's point of view.

The Performance Tree formalism incorporates a wide range of operators, which allow performance queries to express concepts used in classical stochastic property verification and to address quantitative performance measures of relevance. Currently, Performance Trees are able to reason about passage time densities and distributions, their convolutions, moments and percentiles, steady-state distributions, transient probability measures, sets of states that satisfy certain steady-state or transient probability constraints, mean rates of occurrence of actions, and standard logical and arithmetic operations and comparisons. To the best of our knowledge, Performance Trees are the first query specification formalism to support this level of expressiveness, which can be further extended by using a macro mechanism to define custom performance concepts as combinations of basic Performance Tree operators, or by incorporating new stand-alone operators.

Performance Trees are versatile in terms of the models that they can express queries on. Due to their abstract state and action specification mechanism, Performance Trees can be used with a wide range of modelling formalisms that are based on state-transition systems.

We have introduced the Performance Tree formalism together with a syntax that defines the appropriate use of its operators, a framework that defines operator types and prescribes applicable compatibility requirements, and quantitative semantics that rigorously establish their underlying mathematical meaning.

7.2 Practical Contributions

To enable the use of Performance Trees in real-life analysis scenarios, we have incorporated software support for the formalism into *PIPE2*, a model specification tool that forms part of a parallel and distributed performance analysis environment to provide accessible performance query specification and scalable automated evaluation capabilities.

PIPE2 is an open-source platform-independent Petri net editor, which supports the design and analysis of GSPN-based system models. We have implemented a Performance Tree query editor in the form of a *PIPE2* module to enable the graphical and textual specification of performance queries. This module serves at the same time as a front-end to the analysis environment, which allows users to submit analysis jobs for evaluation, and which visualises obtained results. We have also implemented a natural language-based query translation mechanism, which aids users in the intuitive verification of their queries during the specification process. In addition, a guided natural language-based query specification mechanism that constructs query trees automatically is also available.

Alongside extensions to *PIPE2*, we have developed an integrated analysis environment to enable the fully automated analysis of system models, without requiring manual intervention. The environment consists of a number of different components: *PIPE2*, which provides the graphical interface and single point of access to users; the Analysis Server, which manages and coordinates analysis jobs; a set of specialised parallel and distributed analysis tools, which perform numerical analyses to calculate requested performance measures; and a Grid-based analysis cluster, which provides the supporting computational infrastructure.

To illustrate the applicability of Performance Trees in real-life analysis scenarios, we have presented a number of case studies in which we have specified and evaluated complex performance queries on system models with the help of our analysis environment. We have analysed models of an electronic voting system, an online transaction system and a hospital's Accident & Emergency unit.

7.3 Applications

Performance Trees represent an accessible and versatile alternative to existing approaches to performance query specification. Their broad expressiveness, which combines a wide range of performance concepts from a number of specification formalisms, and their extensibility, which allows them to be adapted to custom analysis requirements and scenarios, makes them an attractive choice and a valuable asset to system engineers.

Performance Trees can be used for the specification of performance queries on stochastic models in scenarios where systems can be represented by state-based stochastic modelling formalisms, such as stochastic Petri nets, stochastic process algebras and closed queueing networks. It is also often necessary to evaluate the efficiency of processes represented as workflows. Since workflows can be modelled stochastically, Performance Trees lend themselves to their analysis very naturally.

Numerous industries depend on the reliable operation and near-optimal performance of systems that support their activities, such as computer and business-specific systems. Manufacturing, for example, is heavily reliant on machinery, whose efficiency and reliability are of strategic importance. Hence, performance analysis is essential. The telecommunications industry, as a further example, is governed by QoS guarantees that are agreed upon with customers in Service Level Agreement contracts. It is therefore important to be able to predict compliance with SLAs. Similarly, financial institutions critically depend on the reliability and performance of their systems (especially databases and trading platforms), and thus consider performance analysis to be of utmost importance. Public health care institutions, furthermore, require response time analyses of patient flow models to help improve patient-perceived QoS amidst ever-growing service demand.

Performance Trees also have the potential to not only provide a way of verifying compliance with SLA-related QoS requirements on models before implementation, but to provide a way to monitor compliance with QoS requirements during system operation. It is possible to extract performance properties from running systems directly by assigning monitoring agents to them, which are self-contained software components that act as passive measurement data collectors. By sampling performance information in this way, instead of deriving it through numerical analysis, results of performance queries represent a significantly more accurate reflection of actual system behaviour. Ambiguity is often a considerable problem in QoS requirement specification, and it is often unclear during analysis what system properties are being observed. Performance Trees have the potential to serve as a unifying framework that supports the (also natural language-based) specification of performance properties on real-life systems and their model representations at the same time. The problem of ambiguity will be overcome by the ability of Performance Trees to define in an unambiguous manner what exactly is being measured. This enables system designers to compare real-life systems with corresponding models in an accessible way in order to locate possible performance bottlenecks.

These are only a few selected examples of areas in which Performance Trees could be used effectively, but they give a good indication of the potential for the application of Performance Trees in other industrial scenarios.

7.4 Future Work

For the near future, we envisage the implementation of analysis tools for the evaluation of the *ProbInStates*, *SS:S*, *StatesAtTime*, *Macro* and $\subseteq$ operators. Although they form part of the set of basic Performance Tree operators, they can presently not be evaluated, due to a lack of tool support. Performance queries that reason about individual tagged customers can already be evaluated manually; however, *PIPE2*'s graphical Performance Tree constraint specification mechanism for sets of states needs to be extended to enable the automated evaluation of queries addressing tagged customers.

The extension of the current level of expressiveness of Performance Trees will also need to be addressed, to cater for analysis scenarios that require the ability to reason about concepts of performance that are currently not supported by existing operators. Such extensions will require minor enhancements of *PIPE2* to support the graphical specification of queries that use the new operators, and supporting analysis tools will need to be integrated into the analysis environment to cater for their evaluation.

To provide users with more flexibility in terms of the models that they can query with Performance Trees, the Analysis Server will be integrated with other front-ends, such as the *PEPA Plug-in* [Tribastone07], for example. Furthermore, in an effort to support global collaboration, our analysis environment will be integrated with *PerformDB* [Argent-Katwala06, Argent-Katwala07a], an online database of performance models and related performance results. This will facilitate the automated storage and retrieval of models specified in *PIPE2*, and enable users to globally share their models and results.

In addition, existing analysis tools will be further enhanced to support the evaluation of performance queries on models of even greater complexity than presently possible. An expansion of the hardware infrastructure that provides the computational resources to our analysis environment will also take place. This will be realised by interlinking our analysis cluster with multiple globally dispersed resources to increase the analysis environment's ability to cope with very large system models that necessitate extremely resource-intensive computations.

Appendix A

Case Study Model Descriptions

A.1 Electronic Voting System

A.1.1 *DNAmaca* Model

The *DNAmaca* model that corresponds to the GSPN model of Figure 6.1 is given below:

```
\model{

  \constant{no_of_voters}{100}
  \constant{no_of_pollers}{10}
  \constant{no_of_servers}{10}

  \constant{rate_t1}{1.0}
  \constant{rate_t2}{1.0}
  \constant{rate_t3}{0.03}
  \constant{rate_t4}{0.2}
  \constant{rate_t5}{0.3}
  \constant{rate_t6}{0.01}
  \constant{rate_t7}{0.2}
  \constant{rate_t8}{0.3}
  \constant{rate_t9}{0.00001}

  \statevector{\type{short}{
    NOT_VOTED, VOTED, POLLING_UNITS, VOTE_TAKEN, SERVERS,
    SERVERS_BROKEN, POLLERS_BROKEN}
  }

  \initial{
    NOT_VOTED = no_of_voters;
    VOTED = 0;
    POLLING_UNITS = no_of_pollers;
    VOTE_TAKEN = 0;
    SERVERS = no_of_servers;
    SERVERS_BROKEN = 0;
    POLLERS_BROKEN = 0;
  }

  \transition{t1}{
    \condition{
          NOT_VOTED > 0 && POLLING_UNITS > 0
    }
    \action{
          next->NOT_VOTED = NOT_VOTED - 1;
          next->VOTED = VOTED + 1;
          next->POLLING_UNITS = POLLING_UNITS - 1;
          next->VOTE_TAKEN = VOTE_TAKEN + 1;
    }
    \rate{
          (NOT_VOTED < POLLING_UNITS) ?
          (rate_t1 * (double)NOT_VOTED) :
          (rate_t1 * (double)POLLING_UNITS)
    }
  }

  \transition{t2}{
    \condition{
          VOTE_TAKEN > 0 && SERVERS > 0
    }
    \action{
          next->VOTE_TAKEN = VOTE_TAKEN - 1;
          next->POLLING_UNITS = POLLING_UNITS + 1;
    }
    \rate{
          rate_t2 * (double)SERVERS
    }
  }

  \transition{t3}{
    \condition{
          POLLING_UNITS > 0
    }
    \action{
          next->POLLING_UNITS = POLLING_UNITS - 1;
          next->POLLERS_BROKEN = POLLERS_BROKEN + 1;
    }
    \rate{
          rate_t3 * (double)POLLING_UNITS
    }
  }

  \transition{t4}{
    \condition{
          POLLERS_BROKEN > 0
    }
    \action{
          next->POLLING_UNITS = POLLING_UNITS + 1;
          next->POLLERS_BROKEN = POLLERS_BROKEN - 1;
    }
    \rate{
          rate_t4 * (double)POLLERS_BROKEN
    }
  }

  \transition{t5}{
    \condition{
          POLLERS_BROKEN > PP-1
    }
    \action{
          next->POLLERS_BROKEN = POLLERS_BROKEN - PP;
          next->POLLING_UNITS = POLLING_UNITS + PP;
    }
    \rate{
          rate_t5
    }
  }

  \transition{t6}{
    \condition{
          SERVERS > 0
    }
    \action{
          next->SERVERS = SERVERS - 1;
          next->SERVERS_BROKEN = SERVERS_BROKEN + 1;
    }
    \rate{
          rate_t6 * (double)SERVERS
    }
  }

  \transition{t7}{
    \condition{
          SERVERS_BROKEN > 0
    }
    \action{
          next->SERVERS_BROKEN = SERVERS_BROKEN - 1;
          next->SERVERS = SERVERS + 1;
    }
    \rate{
          rate_t7 * (double)SERVERS_BROKEN
    }
  }

  \transition{t8}{
    \condition{
          SERVERS_BROKEN > SS-1
    }
    \action{
          next->SERVERS_BROKEN = SERVERS_BROKEN - SS;
          next->SERVERS = SERVERS + SS;
    }
    \rate{
          rate_t8
    }
  }

  \transition{t9}{
    \condition{
          VOTED > VV-1
    }
    \action{
          next->VOTED = VOTED - VV;
          next->NOT_VOTED = NOT_VOTED + VV;
    }
    \rate{
          rate_t9
    }
  }
}
```

A.1.2 PEPA Model

The PEPA model that corresponds to the GSPN model of Figure 6.1 is given below:

```
not_voted = (vote, r1).voted

voted = (reset, r2).not_voted

vote_taken = (process_vote, r3).polling_units

polling_units = (vote, r1).vote_taken +
                (polling_unit_breakdown, r4).pollers_broken

pollers_broken = (polling_unit_repaired, r5).polling_units +
                 (all_polling_units_repaired, r6).polling_units

servers = (process_vote, r3).servers +
          (server_breakdown, r7).servers_broken

servers_broken = (server_repaired, r8).servers +
                 (all_servers_repaired, r9).servers

voters = not_voted[CC]

pollers = polling_units[MM]

voting_servers = servers[NN]

Voting_System = (voters <vote> pollers) <>
                (voting_servers <process_vote> vote_taken)
```

A.2 Online Transaction System Model

A.2.1 *DNAmaca* Model

The *DNAmaca* model that corresponds to the GSPN model of Figure 3.9 is given below:

```
\model{

  \constant{no_of_customers}{8}

  \constant{rate_enter_site}{0.6}
  \constant{rate_quit_site}{0.03}
  \constant{rate_go_elsewhere}{0.8}
  \constant{rate_browse_catalogue}{0.7}
  \constant{rate_quit_browsing}{0.02}
  \constant{rate_select_item}{0.7}
  \constant{rate_jump_to_checkout}{0.1}
  \constant{rate_back_to_browse_from_select}{0.2}
  \constant{rate_quit_selecting}{0.02}
  \constant{rate_go_to_checkout}{0.6}
  \constant{rate_back_to_browse_from_checkout}{0.1}
  \constant{rate_quit_checking_out}{0.01}
  \constant{rate_log_in}{0.3}
  \constant{rate_register}{0.5}
  \constant{rate_provide_address}{0.9}
  \constant{rate_quit_login}{0.01}
  \constant{rate_provide_details}{0.9}
  \constant{rate_quit_registration}{0.01}
  \constant{rate_provide_billing_info}{0.9}
  \constant{rate_quit_address_info_provision}{0.01}
  \constant{rate_confirm_order}{0.9}
  \constant{rate_quit_billing_info_provision}{0.01}
  \constant{rate_back_to_browse_from_confirm}{0.5}
  \constant{rate_leave_site}{0.5}

  \statevector{
    \type{short}{
      NOT_AT_SITE, SITE_ENTERED, BROWSING_CATALOGUE,
      ITEM_SELECTED, AT_CHECKOUT, LOGGED_IN, REGISTERED,
      ADDRESS_PROVIDED, BILLING_INFO_PROVIDED,
      ORDER_CONFIRMED, TRANSACTION_ABORTED, tagged_location
    }
  }

  \initial{
    NOT_AT_SITE = no_of_customers;
    SITE_ENTERED = 0;
    BROWSING_CATALOGUE = 0;
    ITEM_SELECTED = 0;
    AT_CHECKOUT = 0;
    LOGGED_IN = 0;
    REGISTERED = 0;
    ADDRESS_PROVIDED = 0;
    BILLING_INFO_PROVIDED = 0;
    ORDER_CONFIRMED = 0;
    TRANSACTION_ABORTED = 0;
    tagged_location = 0;
  }


  % enter_site

  \transition{T0}{
    \condition{
         (tagged_location != 0 && NOT_AT_SITE > 0) ||
         (tagged_location == 0 && NOT_AT_SITE > 1)
    }
    \action{
          next->NOT_AT_SITE = NOT_AT_SITE - 1;
          next->SITE_ENTERED = SITE_ENTERED + 1;
    }
    \rate{
          (tagged_location == 0)
          ?
          (rate_enter_site * ((double)(NOT_AT_SITE - 1)) *
          (((double)(NOT_AT_SITE - 1)) / ((double)NOT_AT_SITE)))
          :
          (rate_enter_site * ((double)(NOT_AT_SITE)))
    }
  }

  \transition{T0_tagged}{
    \condition{
          (tagged_location == 0) && (NOT_AT_SITE > 0)
    }
    \action{
          next->NOT_AT_SITE = NOT_AT_SITE - 1;
          next->SITE_ENTERED = SITE_ENTERED + 1;
          next->tagged_location = 1;
    }
    \rate{
          NOT_AT_SITE > 1
          ?
          (rate_enter_site * ((double)(1)) * (((double)(1)) /
          ((double)NOT_AT_SITE)))
          :
          rate_enter_site
    }
  }

  % quit_site

  \transition{T1}{
    \condition{
          (tagged_location != 1 && SITE_ENTERED > 0) ||
          (tagged_location == 1 && SITE_ENTERED > 1)
    }
    \action{
          next->SITE_ENTERED = SITE_ENTERED - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
    }
    \rate{
          (tagged_location == 1)
          ?
          (rate_quit_site * ((double)(SITE_ENTERED - 1)) *
          (((double)(SITE_ENTERED - 1)) / ((double)SITE_ENTERED)))
          :
          (rate_quit_site * ((double)(SITE_ENTERED)))
    }
  }

  \transition{T1_tagged}{
    \condition{
          (tagged_location == 1) && (SITE_ENTERED > 0)
    }
    \action{
          next->SITE_ENTERED = SITE_ENTERED - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
          next->tagged_location = 10;
    }
    \rate{
          SITE_ENTERED > 1
          ?
          (rate_quit_site * ((double)(1)) * (((double)(1)) /
          ((double)SITE_ENTERED)))
          :
          rate_quit_site
    }
  }

  % browse_catalogue

  \transition{T2}{
    \condition{
          (tagged_location != 1 && SITE_ENTERED > 0) ||
          (tagged_location == 1 && SITE_ENTERED > 1)
    }
    \action{
          next->SITE_ENTERED = SITE_ENTERED - 1;
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
    }
   \rate{
          (tagged_location == 1)
          ?
          (rate_browse_catalogue * ((double)(SITE_ENTERED - 1)) *
          (((double)(SITE_ENTERED - 1)) / ((double)SITE_ENTERED)))
          :
          (rate_browse_catalogue * ((double)(SITE_ENTERED)))
    }
  }
```

```
\transition{T2_tagged}{
  \condition{
        (tagged_location == 1) && (SITE_ENTERED > 0)
  }
  \action{
        next->SITE_ENTERED = SITE_ENTERED - 1;
        next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
        next->tagged_location = 2;
  }
  \rate{
        SITE_ENTERED > 1
        ?
        (rate_browse_catalogue * ((double)(1)) * (((double)(1)) /
        ((double)SITE_ENTERED)))
        :
        rate_browse_catalogue
  }
}

% select_item

\transition{T3}{
  \condition{
        (tagged_location != 2 && BROWSING_CATALOGUE > 0) ||
        (tagged_location == 2 && BROWSING_CATALOGUE > 1)
  }
  \action{
        next->BROWSING_CATALOGUE = BROWSING_CATALOGUE - 1;
        next->ITEM_SELECTED = ITEM_SELECTED + 1;
  }
  \rate{
        (tagged_location == 2)
        ?
        (rate_select_item * ((double)(BROWSING_CATALOGUE - 1)) *
        (((double)(BROWSING_CATALOGUE - 1)) /
        ((double)BROWSING_CATALOGUE)))
        :
        (rate_select_item * ((double)(BROWSING_CATALOGUE)))
  }
}

\transition{T3_tagged}{
  \condition{
        (tagged_location == 2) && (BROWSING_CATALOGUE > 0)
  }
  \action{
        next->BROWSING_CATALOGUE = BROWSING_CATALOGUE - 1;
        next->ITEM_SELECTED = ITEM_SELECTED + 1;
        next->tagged_location = 3;
  }
  \rate{
        BROWSING_CATALOGUE > 1
        ?
        (rate_select_item * ((double)(1)) * (((double)(1)) /
        ((double)BROWSING_CATALOGUE)))
        :
```

```
        rate_select_item
  }
}

% go_to_checkout

\transition{T4}{
  \condition{
        (tagged_location != 3 && ITEM_SELECTED > 0) ||
        (tagged_location == 3 && ITEM_SELECTED > 1)
  }
  \action{
        next->ITEM_SELECTED = ITEM_SELECTED - 1;
        next->AT_CHECKOUT = AT_CHECKOUT + 1;
  }
  \rate{
        (tagged_location == 3)
        ?
        (rate_go_to_checkout * ((double)(ITEM_SELECTED - 1)) *
        (((double)(ITEM_SELECTED - 1)) / ((double)ITEM_SELECTED)))
        :
        (rate_go_to_checkout * ((double)(ITEM_SELECTED)))
  }
}

\transition{T4_tagged}{
  \condition{
        (tagged_location == 3) && (ITEM_SELECTED > 0)
  }
  \action{
        next->ITEM_SELECTED = ITEM_SELECTED - 1;
        next->AT_CHECKOUT = AT_CHECKOUT + 1;
        next->tagged_location = 4;
  }
  \rate{
        ITEM_SELECTED > 1
        ?
        (rate_go_to_checkout * ((double)(1)) * (((double)(1)) /
        ((double)ITEM_SELECTED)))
        :
        rate_go_to_checkout
  }
}

% jump_to_checkout

\transition{T5}{
  \condition{
        (tagged_location != 2 && BROWSING_CATALOGUE > 0) ||
        (tagged_location == 2 && BROWSING_CATALOGUE > 1)
  }
  \action{
        next->BROWSING_CATALOGUE = BROWSING_CATALOGUE - 1;
        next->AT_CHECKOUT = AT_CHECKOUT + 1;
  }
```

```
    \rate{
          (tagged_location == 2)
          ?
          (rate_jump_to_checkout * ((double)(BROWSING_CATALOGUE - 1)) *
          (((double)(BROWSING_CATALOGUE - 1)) /
          ((double)BROWSING_CATALOGUE)))
          :
          (rate_jump_to_checkout * ((double)(BROWSING_CATALOGUE)))
    }
  }

  \transition{T5_tagged}{
    \condition{
          (tagged_location == 2) && (BROWSING_CATALOGUE > 0)
    }
    \action{
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE - 1;
          next->AT_CHECKOUT = AT_CHECKOUT + 1;
          next->tagged_location = 4;
    }
    \rate{
          BROWSING_CATALOGUE > 1
          ?
          (rate_jump_to_checkout * ((double)(1)) * (((double)(1)) /
          ((double)BROWSING_CATALOGUE)))
          :
          rate_jump_to_checkout
    }
  }

  % quit_browsing

  \transition{T6}{
    \condition{
          (tagged_location != 2 && BROWSING_CATALOGUE > 0) ||
          (tagged_location == 2 && BROWSING_CATALOGUE > 1)
    }
    \action{
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
    }
    \rate{
          (tagged_location == 2)
          ?
          (rate_quit_browsing * ((double)(BROWSING_CATALOGUE - 1)) *
          (((double)(BROWSING_CATALOGUE - 1)) /
          ((double)BROWSING_CATALOGUE)))
          :
          (rate_quit_browsing * ((double)(BROWSING_CATALOGUE)))
    }
  }

  \transition{T6_tagged}{
    \condition{
          (tagged_location == 2) && (BROWSING_CATALOGUE > 0)
    }
    \action{
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
          next->tagged_location = 10;
    }
    \rate{
          BROWSING_CATALOGUE > 1
          ?
          (rate_quit_browsing * ((double)(1)) * (((double)(1)) /
          ((double)BROWSING_CATALOGUE)))
          :
          rate_quit_browsing
    }
  }

  % back_to_browse_from_select

  \transition{T7}{
    \condition{
          (tagged_location != 3 && ITEM_SELECTED > 0) ||
          (tagged_location == 3 && ITEM_SELECTED > 1)
    }
    \action{
          next->ITEM_SELECTED = ITEM_SELECTED - 1;
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
    }
    \rate{
          (tagged_location == 3)
          ?
          (rate_back_to_browse_from_select *
          ((double)(ITEM_SELECTED - 1)) *
          (((double)(ITEM_SELECTED - 1)) /
          ((double)ITEM_SELECTED)))
          :
          (rate_back_to_browse_from_select *
          ((double)(ITEM_SELECTED)))
    }
  }

  \transition{T7_tagged}{
    \condition{
          (tagged_location == 3) && (ITEM_SELECTED > 0)
    }
    \action{
          next->ITEM_SELECTED = ITEM_SELECTED - 1;
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
          next->tagged_location = 2;
    }
    \rate{
          ITEM_SELECTED > 1
          ?
          (rate_back_to_browse_from_select * ((double)(1)) *
          (((double)(1)) / ((double)ITEM_SELECTED)))
          :
          rate_back_to_browse_from_select
    }
```

```
}

% back_to_browse_from_checkout

\transition{T8}{
  \condition{
        (tagged_location != 4 && AT_CHECKOUT > 0) ||
        (tagged_location == 4 && AT_CHECKOUT > 1)
  }
  \action{
        next->AT_CHECKOUT = AT_CHECKOUT - 1;
        next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
  }
  \rate{
        (tagged_location == 4)
        ?
        (rate_back_to_browse_from_checkout *
        ((double)(AT_CHECKOUT - 1)) *
        (((double)(AT_CHECKOUT - 1)) /
        ((double)AT_CHECKOUT)))
        :
        (rate_back_to_browse_from_checkout *
        ((double)(AT_CHECKOUT)))
  }
}

\transition{T8_tagged}{
  \condition{
        (tagged_location == 4) && (AT_CHECKOUT > 0)
  }
  \action{
        next->AT_CHECKOUT = AT_CHECKOUT - 1;
        next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
        next->tagged_location = 2;
  }
  \rate{
        AT_CHECKOUT > 1
        ?
        (rate_back_to_browse_from_checkout * ((double)(1)) *
        (((double)(1)) / ((double)AT_CHECKOUT)))
        :
        rate_back_to_browse_from_checkout
  }
}

% quit_selecting

\transition{T9}{
  \condition{
        (tagged_location != 3 && ITEM_SELECTED > 0) ||
        (tagged_location== 3 && ITEM_SELECTED > 1)
  }
  \action{
        next->ITEM_SELECTED = ITEM_SELECTED - 1;
        next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
  }
```

```
    \rate{
          (tagged_location == 3)
          ?
          (rate_quit_selecting * ((double)(ITEM_SELECTED - 1)) *
          (((double)(ITEM_SELECTED - 1)) /
          ((double)ITEM_SELECTED)))
          :
          (rate_quit_selecting * ((double)(ITEM_SELECTED)))
    }
  }

  \transition{T9_tagged}{
    \condition{
          (tagged_location==3) && (ITEM_SELECTED > 0)
    }
    \action{
          next->ITEM_SELECTED = ITEM_SELECTED - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
          next->tagged_location = 10;
    }
    \rate{
          ITEM_SELECTED > 1
          ?
          (rate_quit_selecting * ((double)(1)) * (((double)(1)) /
          ((double)ITEM_SELECTED)))
          :
          rate_quit_selecting
    }
  }

  % quit_checking_out

  \transition{T10}{
    \condition{
          (tagged_location != 4 && AT_CHECKOUT > 0) ||
          (tagged_location == 4 && AT_CHECKOUT > 1)
    }
    \action{
          next->AT_CHECKOUT = AT_CHECKOUT - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
    }
    \rate{
          (tagged_location == 4)
          ?
          (rate_quit_checking_out * ((double)(AT_CHECKOUT - 1)) *
          (((double)(AT_CHECKOUT - 1)) / ((double)AT_CHECKOUT)))
          :
          (rate_quit_checking_out * ((double)(AT_CHECKOUT)))
    }
  }

  \transition{T10_tagged}{
    \condition{
          (tagged_location == 4) && (AT_CHECKOUT > 0)
    }
    \action{
          next->AT_CHECKOUT = AT_CHECKOUT - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
          next->tagged_location = 10;
    }
    \rate{
          AT_CHECKOUT > 1
          ?
          (rate_quit_checking_out * ((double)(1)) *
          (((double)(1)) / ((double)AT_CHECKOUT)))
          :
          rate_quit_checking_out
    }
  }

  % log_in

  \transition{T11}{
    \condition{
          (tagged_location != 4 && AT_CHECKOUT > 0) ||
          (tagged_location == 4 && AT_CHECKOUT > 1)
    }
    \action{
          next->AT_CHECKOUT = AT_CHECKOUT - 1;
          next->LOGGED_IN = LOGGED_IN + 1;
    }
    \rate{
          (tagged_location == 4)
          ?
          (rate_log_in * ((double)(AT_CHECKOUT - 1)) *
          (((double)(AT_CHECKOUT - 1)) /
          ((double)AT_CHECKOUT)))
          :
          (rate_log_in * ((double)(AT_CHECKOUT)))
    }
  }

  \transition{T11_tagged}{
    \condition{
          (tagged_location == 4) && (AT_CHECKOUT > 0)
    }
    \action{
          next->AT_CHECKOUT = AT_CHECKOUT - 1;
          next->LOGGED_IN = LOGGED_IN + 1;
          next->tagged_location = 5;
    }
    \rate{
          AT_CHECKOUT > 1
          ?
          (rate_log_in * ((double)(1)) * (((double)(1)) /
          ((double)AT_CHECKOUT)))
          :
          rate_log_in
    }
  }
```

```
% register

\transition{T12}{
  \condition{
        (tagged_location != 4 && AT_CHECKOUT > 0) ||
        (tagged_location == 4 && AT_CHECKOUT > 1)
  }
  \action{
        next->AT_CHECKOUT = AT_CHECKOUT - 1;
        next->REGISTERED = REGISTERED + 1;
  }
  \rate{
        (tagged_location == 4)
        ?
        (rate_register * ((double)(AT_CHECKOUT - 1)) *
        (((double)(AT_CHECKOUT - 1)) /
        ((double)AT_CHECKOUT)))
        :
        (rate_register * ((double)(AT_CHECKOUT)))
  }
}

\transition{T12_tagged}{
  \condition{
        (tagged_location == 4) && (AT_CHECKOUT > 0)
  }
  \action{
        next->AT_CHECKOUT = AT_CHECKOUT - 1;
        next->REGISTERED = REGISTERED + 1;
        next->tagged_location = 6;
  }
  \rate{
        AT_CHECKOUT > 1
        ?
        (rate_register * ((double)(1)) * (((double)(1)) /
        ((double)AT_CHECKOUT)))
        :
        rate_register
  }
}

% provide_address

\transition{T13}{
  \condition{
        (tagged_location != 5 && LOGGED_IN > 0) ||
        (tagged_location == 5 && LOGGED_IN > 1)
  }
  \action{
        next->LOGGED_IN = LOGGED_IN - 1;
        next->ADDRESS_PROVIDED = ADDRESS_PROVIDED + 1;
  }
  \rate{
        (tagged_location == 5)
        ?
        (rate_provide_address * ((double)(LOGGED_IN - 1)) *
        (((double)(LOGGED_IN - 1)) / ((double)LOGGED_IN)))
        :
        (rate_provide_address * ((double)(LOGGED_IN)))
  }
}

\transition{T13_tagged}{
  \condition{
        (tagged_location == 5) && (LOGGED_IN > 0)
  }
  \action{
        next->LOGGED_IN = LOGGED_IN - 1;
        next->ADDRESS_PROVIDED = ADDRESS_PROVIDED + 1;
        next->tagged_location = 7;
  }
  \rate{
        LOGGED_IN > 1
        ?
        (rate_provide_address * ((double)(1)) *
        (((double)(1)) / ((double)LOGGED_IN)))
        :
        rate_provide_address
  }
}

% provide_details

\transition{T14}{
  \condition{
        (tagged_location != 6 && REGISTERED > 0) ||
        (tagged_location == 6 && REGISTERED > 1)
  }
  \action{
        next->REGISTERED = REGISTERED - 1;
        next->ADDRESS_PROVIDED = ADDRESS_PROVIDED + 1;
  }
  \rate{
        (tagged_location == 6)
        ?
        (rate_provide_details * ((double)(REGISTERED - 1)) *
        (((double)(REGISTERED - 1)) / ((double)REGISTERED)))
        :
        (rate_provide_details * ((double)(REGISTERED)))
  }
}

\transition{T14_tagged}{
  \condition{
        (tagged_location == 6) && (REGISTERED > 0)
  }
  \action{
        next->REGISTERED = REGISTERED - 1;
        next->ADDRESS_PROVIDED = ADDRESS_PROVIDED + 1;
        next->tagged_location = 7;
  }
  \rate{
        REGISTERED > 1
        ?
        (rate_provide_details * ((double)(1)) *
        (((double)(1)) / ((double)REGISTERED)))
        :
        rate_provide_details
  }
}

% quit_registration

\transition{T15}{
  \condition{
        (tagged_location != 6 && REGISTERED > 0) ||
        (tagged_location == 6 && REGISTERED > 1)
  }
  \action{
        next->REGISTERED = REGISTERED - 1;
        next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
  }
  \rate{
        (tagged_location == 6)
        ?
        (rate_quit_registration * ((double)(REGISTERED - 1)) *
        (((double)(REGISTERED - 1)) / ((double)REGISTERED)))
        :
        (rate_quit_registration * ((double)(REGISTERED)))
  }
}

\transition{T15_tagged}{
  \condition{
        (tagged_location == 6) && (REGISTERED > 0)
  }
  \action{
        next->REGISTERED = REGISTERED - 1;
        next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
        next->tagged_location = 10;
  }
  \rate{
        REGISTERED > 1
        ?
        (rate_quit_registration * ((double)(1)) *
        (((double)(1)) / ((double)REGISTERED)))
        :
        rate_quit_registration
  }
}

% quit_login

\transition{T16}{
  \condition{
        (tagged_location != 5 && LOGGED_IN > 0) ||
        (tagged_location == 5 && LOGGED_IN > 1)
```

```
    }
    \action{
          next->LOGGED_IN = LOGGED_IN - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
    }
    \rate{
          (tagged_location == 5)
          ?
          (rate_quit_login * ((double)(LOGGED_IN - 1)) *
          (((double)(LOGGED_IN - 1)) / ((double)LOGGED_IN)))
          :
          (rate_quit_login * ((double)(LOGGED_IN)))
    }
  }

  \transition{T16_tagged}{
    \condition{
          (tagged_location == 5) && (LOGGED_IN > 0)
    }
    \action{
          next->LOGGED_IN = LOGGED_IN - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
          next->tagged_location = 10;
    }
    \rate{
          LOGGED_IN > 1
          ?
          (rate_quit_login * ((double)(1)) *
          (((double)(1)) / ((double)LOGGED_IN)))
          :
          rate_quit_login
    }
  }

  % quit_address_info_provision

  \transition{T17}{
    \condition{
          (tagged_location != 7 && ADDRESS_PROVIDED > 0) ||
          (tagged_location == 7 && ADDRESS_PROVIDED > 1)
    }
    \action{
          next->ADDRESS_PROVIDED = ADDRESS_PROVIDED - 1;
          next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
    }
    \rate{
          (tagged_location == 7)
          ?
          (rate_quit_address_info_provision *
          ((double)(ADDRESS_PROVIDED - 1)) *
          (((double)(ADDRESS_PROVIDED - 1)) /
          ((double)ADDRESS_PROVIDED)))
          :
          (rate_quit_address_info_provision *
          ((double)(ADDRESS_PROVIDED)))
    }
```

```
}

\transition{T17_tagged}{
  \condition{
        (tagged_location == 7) && (ADDRESS_PROVIDED > 0)
  }
  \action{
        next->ADDRESS_PROVIDED = ADDRESS_PROVIDED - 1;
        next->TRANSACTION_ABORTED = TRANSACTION_ABORTED + 1;
        next->tagged_location = 10;
  }
  \rate{
        ADDRESS_PROVIDED > 1
        ?
        (rate_quit_address_info_provision * ((double)(1)) *
        (((double)(1)) / ((double)ADDRESS_PROVIDED)))
        :
        rate_quit_address_info_provision
  }
}

% provide_billing_info

\transition{T18}{
  \condition{
        (tagged_location != 7 && ADDRESS_PROVIDED > 0) ||
        (tagged_location == 7 && ADDRESS_PROVIDED > 1)
  }
  \action{
        next->ADDRESS_PROVIDED = ADDRESS_PROVIDED - 1;
        next->BILLING_INFO_PROVIDED =
              BILLING_INFO_PROVIDED + 1;
  }
  \rate{
        (tagged_location == 7)
        ?
        (rate_provide_billing_info *
        ((double)(ADDRESS_PROVIDED - 1)) *
        (((double)(ADDRESS_PROVIDED - 1)) /
        ((double)ADDRESS_PROVIDED)))
        :
        (rate_provide_billing_info *
        ((double)(ADDRESS_PROVIDED)))
  }
}

\transition{T18_tagged}{
  \condition{
        (tagged_location == 7) && (ADDRESS_PROVIDED > 0)
  }
  \action{
        next->ADDRESS_PROVIDED = ADDRESS_PROVIDED - 1;
        next->BILLING_INFO_PROVIDED =
              BILLING_INFO_PROVIDED + 1;
        next->tagged_location = 8;
  }
```

```
    \rate{
          ADDRESS_PROVIDED > 1
          ?
          (rate_provide_billing_info * ((double)(1)) *
          (((double)(1)) / ((double)ADDRESS_PROVIDED)))
          :
          rate_provide_billing_info
    }
  }

  % confirm_order

  \transition{T19}{
    \condition{
          (tagged_location != 8 && BILLING_INFO_PROVIDED > 0) ||
          (tagged_location == 8 && BILLING_INFO_PROVIDED > 1)
    }
    \action{
          next->BILLING_INFO_PROVIDED =
                BILLING_INFO_PROVIDED - 1;
          next->ORDER_CONFIRMED =
                ORDER_CONFIRMED + 1;
    }
    \rate{
          (tagged_location == 8)
          ?
          (rate_confirm_order *
          ((double)(BILLING_INFO_PROVIDED - 1)) *
          (((double)(BILLING_INFO_PROVIDED - 1)) /
          ((double)BILLING_INFO_PROVIDED)))
          :
          (rate_confirm_order *
          ((double)(BILLING_INFO_PROVIDED)))
    }
  }

  \transition{T19_tagged}{
    \condition{
          (tagged_location == 8) && (BILLING_INFO_PROVIDED > 0)
    }
    \action{
          next->BILLING_INFO_PROVIDED =
                BILLING_INFO_PROVIDED - 1;
          next->ORDER_CONFIRMED =
                ORDER_CONFIRMED + 1;
          next->tagged_location = 9;
    }
    \rate{
          BILLING_INFO_PROVIDED > 1
          ?
          (rate_confirm_order * ((double)(1)) *
          (((double)(1)) / ((double)BILLING_INFO_PROVIDED)))
          :
          rate_confirm_order
    }
  }
```

```
% quit_billing_info_provision

\transition{T20}{
  \condition{
        (tagged_location != 8 && BILLING_INFO_PROVIDED > 0) ||
        (tagged_location == 8 && BILLING_INFO_PROVIDED > 1)
  }
  \action{
        next->BILLING_INFO_PROVIDED =
              BILLING_INFO_PROVIDED - 1;
        next->TRANSACTION_ABORTED =
              TRANSACTION_ABORTED + 1;
  }
  \rate{
        (tagged_location == 8)
        ?
        (rate_quit_billing_info_provision *
        ((double)(BILLING_INFO_PROVIDED - 1)) *
        (((double)(BILLING_INFO_PROVIDED - 1)) /
        ((double)BILLING_INFO_PROVIDED)))
        :
        (rate_quit_billing_info_provision *
        ((double)(BILLING_INFO_PROVIDED)))
  }
}

\transition{T20_tagged}{
  \condition{
        (tagged_location == 8) && (BILLING_INFO_PROVIDED > 0)
  }
  \action{
        next->BILLING_INFO_PROVIDED =
              BILLING_INFO_PROVIDED - 1;
        next->TRANSACTION_ABORTED =
              TRANSACTION_ABORTED + 1;
        next->tagged_location = 10;
  }
  \rate{
        BILLING_INFO_PROVIDED > 1
        ?
        (rate_quit_billing_info_provision * ((double)(1)) *
        (((double)(1)) / ((double)BILLING_INFO_PROVIDED)))
        :
        rate_quit_billing_info_provision
  }
}

% back_to_browse_from_confirm

\transition{T21}{
  \condition{
        (tagged_location != 9 && ORDER_CONFIRMED > 0) ||
        (tagged_location == 9 && ORDER_CONFIRMED > 1)
  }
```

```
    \action{
          next->ORDER_CONFIRMED = ORDER_CONFIRMED - 1;
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
    }
    \rate{
          (tagged_location == 9)
          ?
          (rate_back_to_browse_from_confirm *
          ((double)(ORDER_CONFIRMED - 1)) *
          (((double)(ORDER_CONFIRMED - 1)) /
          ((double)ORDER_CONFIRMED)))
          :
          (rate_back_to_browse_from_confirm *
          ((double)(ORDER_CONFIRMED)))
    }
  }

  \transition{T21_tagged}{
    \condition{
          (tagged_location == 9) && (ORDER_CONFIRMED > 0)
    }
    \action{
          next->ORDER_CONFIRMED = ORDER_CONFIRMED - 1;
          next->BROWSING_CATALOGUE = BROWSING_CATALOGUE + 1;
          next->tagged_location = 2;
    }
    \rate{
          ORDER_CONFIRMED > 1
          ?
          (rate_back_to_browse_from_confirm * ((double)(1)) *
          (((double)(1)) / ((double)ORDER_CONFIRMED)))
          :
          rate_back_to_browse_from_confirm
    }
  }

  % leave_site

  \transition{T22}{
    \condition{
          (tagged_location != 9 && ORDER_CONFIRMED > 0) ||
          (tagged_location == 9 && ORDER_CONFIRMED > 1)
    }
    \action{
          next->ORDER_CONFIRMED = ORDER_CONFIRMED - 1;
          next->NOT_AT_SITE = NOT_AT_SITE + 1;
    }
    \rate{
          (tagged_location == 9)
          ?
          (rate_leave_site * ((double)(ORDER_CONFIRMED - 1)) *
          (((double)(ORDER_CONFIRMED - 1)) /
          ((double)ORDER_CONFIRMED)))
          :
          (rate_leave_site * ((double)(ORDER_CONFIRMED)))
    }
```

```
}

\transition{T22_tagged}{
  \condition{
        (tagged_location == 9) && (ORDER_CONFIRMED > 0)
  }
  \action{
        next->ORDER_CONFIRMED = ORDER_CONFIRMED - 1;
        next->NOT_AT_SITE = NOT_AT_SITE + 1;
        next->tagged_location = 0;
  }
  \rate{
        ORDER_CONFIRMED > 1
        ?
        (rate_leave_site * ((double)(1)) * (((double)(1)) /
        ((double)ORDER_CONFIRMED)))
        :
        rate_leave_site
  }
}

% go_elsewhere

\transition{T23}{
  \condition{
        (tagged_location != 10 && TRANSACTION_ABORTED > 0) ||
        (tagged_location == 10 && TRANSACTION_ABORTED > 1)
  }
  \action{
        next->TRANSACTION_ABORTED = TRANSACTION_ABORTED - 1;
        next->NOT_AT_SITE = NOT_AT_SITE + 1;
  }
  \rate{
        (tagged_location == 10)
        ?
        (rate_go_elsewhere * ((double)(TRANSACTION_ABORTED - 1)) *
        (((double)(TRANSACTION_ABORTED - 1)) /
        ((double)TRANSACTION_ABORTED)))
        :
        (rate_go_elsewhere * ((double)(TRANSACTION_ABORTED)))
  }
}

\transition{T23_tagged}{
  \condition{
        (tagged_location == 10) && (TRANSACTION_ABORTED > 0)
  }
  \action{
        next->TRANSACTION_ABORTED = TRANSACTION_ABORTED - 1;
        next->NOT_AT_SITE = NOT_AT_SITE + 1;
        next->tagged_location = 0;
  }
  \rate{
        TRANSACTION_ABORTED > 1
        ?
        (rate_go_elsewhere * ((double)(1)) * (((double)(1)) /
          ((double)TRANSACTION_ABORTED)))
          :
          rate_go_elsewhere
    }
  }
}
```

A.2.2 PEPA Model

The PEPA model that corresponds to the GSPN model of Figure 3.9 is given below:

```
not_at_site = (enter_site, r1).site_entered

site_entered = (browse_catalogue, r2).browsing_catalogue +
               (quit_site, r3).not_at_site

browsing_catalogue = (select_item, r4).item_selected +
                     (jump_to_checkout, r5).at_checkout +
                     (quit_browsing, r6).not_at_site

item_selected = (go_to_checkout, r7).at_checkout +
                (back_to_browse_from_select, r8).browsing_catalogue +
                (quit_selecting, r9).not_at_site

at_checkout = (log_in, r10).logged_in +
              (register, r11).registered +
              (back_to_browse_from_checkout, r12).browsing_catalogue +
              (quit_checking_out, r13).not_at_site

logged_in = (provide_address, r14).address_provided +
            (quit_login, r15).not_at_site

registered = (provide_details, r16).address_provided +
             (quit_registration, r17).not_at_site

address_provided = (provide_billing_info, r18).billing_info_provided +
                   (quit_address_info_provision, r19).not_at_site

billing_info_provided = (confirm_order, r20).order_confirmed +
                        (quit_billing_info_provision, r21).not_at_site

order_confirmed = (back_to_browse_from_confirm, r22).browsing_catalogue +
                  (quit_order_confirmation, r23).not_at_site

OTS = not_at_site[CC]
```

A.3 Hospital Accident & Emergency Unit Model

A.3.1 *DNAmaca* Model

The *DNAmaca* model that corresponds to the GSPN model of Figure 6.22 is given below:

```
\model{

  \constant{no_of_people}{8}
  \constant{no_of_nurses}{2}
  \constant{no_of_doctors}{2}
  \constant{no_of_ambulances}{1}

  \constant{rate_fall_ill}{0.2}
  \constant{rate_walk_in_arrival}{0.3}
  \constant{rate_emergency_call}{0.6}
  \constant{rate_load_patient}{0.6}
  \constant{rate_ambulance_arrival}{0.6}
  \constant{rate_see_nurse}{0.3}
  \constant{rate_see_emergency_nurse}{0.6}
  \constant{rate_complete_assessment}{0.3}
  \constant{rate_complete_emergency_assessment}{0.6}
  \constant{rate_to_doctor}{0.3}
  \constant{rate_see_doctor}{0.3}
  \constant{rate_discharge_treated_patient}{0.3}
  \constant{rate_to_surgery}{0.2}
  \constant{rate_surgery}{0.2}
  \constant{rate_recover}{0.2}
  \constant{rate_discharge_recovered_patient}{0.2}
  \constant{rate_to_tests}{0.4}
  \constant{rate_perform_lab_tests}{0.4}
  \constant{rate_evaluate_results}{0.1}

  \statevector{
    \type{short}{
      WAITING_ROOM, TROLLEY, PATIENT_RECOVERED, HEALTHY, ILL,
      AWAITING_AMBULANCE, IN_TRANSIT, AMBULANCES, PATIENT_ASSESSED,
      NURSES, AMBULANCE_PATIENT_ASSESSED, ASSESSED_PATIENTS,
      TEST_DONE, TREATED_BY_DOCTOR, DOCTORS, SURGERY_DONE,
      WAITING_FOR_DOCTOR, WAITING_FOR_SURGERY, WAITING_FOR_TESTS,
      tagged_location
    }
  }

  \initial{
    WAITING_ROOM = 0;
    TROLLEY = 0;
    PATIENT_RECOVERED = 0;
    HEALTHY = no_of_people;
    ILL = 0;
    AWAITING_AMBULANCE = 0;
    IN_TRANSIT = 0;
    AMBULANCES = no_of_ambulances;
    PATIENT_ASSESSED = 0;
    NURSES = no_of_nurses;
    AMBULANCE_PATIENT_ASSESSED = 0;
    ASSESSED_PATIENTS = 0;
    TEST_DONE = 0;
    TREATED_BY_DOCTOR = 0;
    DOCTORS = no_of_doctors;
    SURGERY_DONE = 0;
    WAITING_FOR_DOCTOR = 0;
    WAITING_FOR_SURGERY = 0;
    WAITING_FOR_TESTS = 0;
    tagged_location = 3;
  }

  %% see nurse %%

  \transition{T0}{
    \condition{
          (WAITING_ROOM > 0 && NURSES > 0 && tagged_location != 0) ||
          (WAITING_ROOM > 1 && NURSES > 0 && tagged_location == 0)
    }
    \action{
          next->WAITING_ROOM = WAITING_ROOM - 1;
          next->NURSES = NURSES - 1;
          next->PATIENT_ASSESSED = PATIENT_ASSESSED + 1;
    }
    \rate{
          (tagged_location == 0)
          ?
          ((WAITING_ROOM < NURSES) ?
          (rate_see_nurse * ((double)(WAITING_ROOM - 1)) *
          (((double)(WAITING_ROOM - 1)) / ((double)WAITING_ROOM))) :
          (rate_see_nurse * ((double)(NURSES)) *
          (((double)(WAITING_ROOM - 1)) / ((double)WAITING_ROOM))))
          :
          ((WAITING_ROOM < NURSES) ? (rate_see_nurse *
          ((double)(WAITING_ROOM))) :
          (rate_see_nurse * ((double)NURSES)))
    }
  }

  \transition{T0_tagged}{
    \condition{
          (WAITING_ROOM > 0 && NURSES > 0) && (tagged_location == 0)
    }
    \action{
          next->WAITING_ROOM = WAITING_ROOM - 1;
          next->NURSES = NURSES - 1;
          next->PATIENT_ASSESSED = PATIENT_ASSESSED + 1;
          next->tagged_location = 8;
    }
    \rate{
          WAITING_ROOM > 1
          ?
          (rate_see_nurse * ((double)(1)) * (((double)(1)) /
          ((double)WAITING_ROOM)))
          :
          rate_see_nurse
  }
}

%% walk-in arrival %%

\transition{T1}{
  \condition{
          (ILL > 0 && tagged_location != 4) ||
          (ILL > 1 && tagged_location == 4)
  }
  \action{
          next->ILL = ILL - 1;
          next->WAITING_ROOM = WAITING_ROOM + 1;
  }
  \weight{
          3.0
  }
}

\transition{T1_tagged}{
  \condition{
         (ILL > 0) && (tagged_location == 4)
  }
  \action{
          next->ILL = ILL - 1;
          next->WAITING_ROOM = WAITING_ROOM + 1;
          next->tagged_location=0;
  }
  \weight{
          3.0
  }
}

%% recover %%

\transition{T10}{
  \condition{
          (SURGERY_DONE > 0 && tagged_location != 15) ||
          (SURGERY_DONE > 1 && tagged_location == 15)
  }
  \action{
          next->SURGERY_DONE = SURGERY_DONE - 1;
          next->PATIENT_RECOVERED = PATIENT_RECOVERED + 1;
          next->DOCTORS = DOCTORS + 1;
  }
  \rate{
          (tagged_location == 15)
          ?
          (rate_recover * ((double)(SURGERY_DONE - 1)) *
          (((double)(SURGERY_DONE - 1)) / ((double)SURGERY_DONE)))
          :
          (rate_recover * ((double)(SURGERY_DONE)))
  }
```

```
}

\transition{T10_tagged}{
  \condition{
        (SURGERY_DONE > 0) && (tagged_location == 15)
  }
  \action{
        next->SURGERY_DONE = SURGERY_DONE - 1;
        next->PATIENT_RECOVERED = PATIENT_RECOVERED + 1;
        next->DOCTORS = DOCTORS + 1;
        next->tagged_location = 2;
  }
  \rate{
        SURGERY_DONE > 1
        ?
        (rate_recover * ((double)(1)) * (((double)(1)) /
        ((double)SURGERY_DONE)))
        :
        rate_recover
  }
}

%% evaluate results %%

\transition{T11}{
  \condition{
        (TEST_DONE > 0 && tagged_location != 12) ||
        (TEST_DONE > 1 && tagged_location == 12)
  }
  \action{
        next->TEST_DONE = TEST_DONE - 1;
        next->ASSESSED_PATIENTS = ASSESSED_PATIENTS + 1;
  }
  \rate{
        (tagged_location == 12)
        ?
        (rate_evaluate_results * ((double)(TEST_DONE - 1)) *
        (((double)(TEST_DONE - 1)) / ((double)TEST_DONE)))
        :
        (rate_evaluate_results * ((double)(TEST_DONE)))
  }
}

\transition{T11_tagged}{
  \condition{
        (TEST_DONE > 0) && (tagged_location == 12)
  }
  \action{
        next->TEST_DONE = TEST_DONE - 1;
        next->ASSESSED_PATIENTS = ASSESSED_PATIENTS + 1;
        next->tagged_location=11;
  }
  \rate{
        TEST_DONE > 1
        ?
        (rate_evaluate_results * ((double)(1)) * (((double)(1)) /
        ((double)TEST_DONE)))
        :
        rate_evaluate_results
  }
}

%% discharge recovered patient %%

\transition{T12}{
  \condition{
        (PATIENT_RECOVERED > 0 && tagged_location != 2) ||
        (PATIENT_RECOVERED > 1 && tagged_location == 2)
  }
  \action{
        next->PATIENT_RECOVERED = PATIENT_RECOVERED - 1;
        next->HEALTHY = HEALTHY + 1;
  }
  \rate{
        (tagged_location == 2)
        ?
        (rate_discharge_recovered_patient *
        ((double)(PATIENT_RECOVERED - 1)) *
        (((double)(PATIENT_RECOVERED - 1)) /
        ((double)PATIENT_RECOVERED)))
        :
        (rate_discharge_recovered_patient *
        ((double)(PATIENT_RECOVERED)))
  }
}

\transition{T12_tagged}{
  \condition{
        (PATIENT_RECOVERED > 0) && (tagged_location==2)
  }
  \action{
        next->PATIENT_RECOVERED = PATIENT_RECOVERED - 1;
        next->HEALTHY = HEALTHY + 1;
        next->tagged_location=3;
  }
  \rate{
        PATIENT_RECOVERED > 1
        ?
        (rate_discharge_recovered_patient * ((double)(1)) *
        (((double)(1)) / ((double)PATIENT_RECOVERED)))
        :
        rate_discharge_recovered_patient
  }
}

%% fall ill %%

\transition{T13}{
  \condition{
        (HEALTHY > 0 && tagged_location != 3) ||
        (HEALTHY > 1 && tagged_location == 3)
  }
```

```
    \action{
          next->HEALTHY = HEALTHY - 1;
          next->ILL = ILL + 1;
    }
    \rate{
          (tagged_location == 3)
          ?
          (rate_fall_ill * ((double)(HEALTHY - 1)) *
          (((double)(HEALTHY - 1)) / ((double)HEALTHY)))
          :
          (rate_fall_ill * ((double)(HEALTHY)))
    }
  }

  \transition{T13_tagged}{
    \condition{
          (HEALTHY > 0) && (tagged_location == 3)
    }
    \action{
          next->HEALTHY = HEALTHY - 1;
          next->ILL = ILL + 1;
          next->tagged_location = 4;
    }
    \rate{
          HEALTHY > 1
          ?
          (rate_fall_ill * ((double)(1)) * (((double)(1)) /
          ((double)HEALTHY)))
          :
          rate_fall_ill
    }
  }

  %% emergency call %%

  \transition{T14}{
    \condition{
          (ILL > 0 && tagged_location != 4) ||
          (ILL > 1 && tagged_location == 4)}
    \action{
          next->ILL = ILL - 1;
          next->AWAITING_AMBULANCE = AWAITING_AMBULANCE + 1;
    }
    \weight{
          6.0
    }
  }

  \transition{T14_tagged}{
    \condition{
          (ILL > 0) && (tagged_location==4)
    }
    \action{
          next->ILL = ILL - 1;
          next->AWAITING_AMBULANCE = AWAITING_AMBULANCE + 1;
          next->tagged_location=5;
    }
    \weight{
          6.0
    }
  }

  %% load patient %%

  \transition{T15}{
    \condition{
          (AWAITING_AMBULANCE > 0 && AMBULANCES > 0 &&
          tagged_location != 5) || (AWAITING_AMBULANCE > 1 &&
          AMBULANCES > 0 && tagged_location == 5)}
    \action{
          next->AWAITING_AMBULANCE = AWAITING_AMBULANCE - 1;
          next->AMBULANCES = AMBULANCES - 1;
          next->IN_TRANSIT = IN_TRANSIT + 1;
    }
    \rate{
          (tagged_location == 5)
          ?
          (rate_load_patient * ((double)(AWAITING_AMBULANCE - 1)) *
          (((double)(AWAITING_AMBULANCE - 1)) /
          ((double)AWAITING_AMBULANCE)))
          :
          (rate_load_patient * ((double)(AWAITING_AMBULANCE)))
    }
  }

  \transition{T15_tagged}{
    \condition{
          (AWAITING_AMBULANCE > 0 && AMBULANCES > 0) &&
          (tagged_location == 5)
    }
    \action{
          next->AWAITING_AMBULANCE = AWAITING_AMBULANCE - 1;
          next->AMBULANCES = AMBULANCES - 1;
          next->IN_TRANSIT = IN_TRANSIT + 1;
          next->tagged_location = 6;
    }
    \rate{
          AWAITING_AMBULANCE > 1
          ?
          (rate_load_patient * ((double)(1)) * (((double)(1)) /
          ((double)AWAITING_AMBULANCE)))
          :
          rate_load_patient
    }
  }

  %% ambulance arrival %%

  \transition{T2}{
    \condition{
          (IN_TRANSIT > 0 && tagged_location != 6) ||
          (IN_TRANSIT > 1 && tagged_location == 6)
    }
    \action{
          next->IN_TRANSIT = IN_TRANSIT - 1;
          next->TROLLEY = TROLLEY + 1;
          next->AMBULANCES = AMBULANCES + 1;
    }
    \rate{
          (tagged_location == 6)
          ?
          (rate_ambulance_arrival * ((double)(IN_TRANSIT - 1)) *
          (((double)(IN_TRANSIT - 1)) /
          ((double)IN_TRANSIT)))
          :
          (rate_ambulance_arrival * ((double)(IN_TRANSIT)))
    }
  }

  \transition{T2_tagged}{
    \condition{
          (IN_TRANSIT > 0) && (tagged_location==6)
    }
    \action{
          next->IN_TRANSIT = IN_TRANSIT - 1;
          next->TROLLEY = TROLLEY + 1;
          next->AMBULANCES = AMBULANCES + 1;
          next->tagged_location=1;
    }
    \rate{
          IN_TRANSIT > 1
          ?
          (rate_ambulance_arrival * ((double)(1)) * (((double)(1)) /
          ((double)IN_TRANSIT)))
          :
          rate_ambulance_arrival
    }
  }

  %% see emergency nurse %%

  \transition{T3}{
    \condition{
          (TROLLEY > 0 && NURSES > 0 && tagged_location != 1) ||
          (TROLLEY > 1 && NURSES > 0 && tagged_location == 1)
    }
    \action{
          next->TROLLEY = TROLLEY - 1;
          next->NURSES = NURSES - 1;
          next->AMBULANCE_PATIENT_ASSESSED =
                AMBULANCE_PATIENT_ASSESSED + 1;
    }
    \rate{
          (tagged_location == 1)
          ?
          ((TROLLEY < NURSES) ?
          (rate_see_emergency_nurse * ((double)(TROLLEY - 1)) *
          (((double)(TROLLEY - 1)) / ((double)TROLLEY))) :
```

```
        (rate_see_emergency_nurse * ((double)(NURSES)) *
        (((double)(TROLLEY - 1)) / ((double)TROLLEY))))
        :
        ((TROLLEY < NURSES) ? (rate_see_emergency_nurse *
        ((double)(TROLLEY))) :
        (rate_see_emergency_nurse * ((double)NURSES)))
  }
}

\transition{T3_tagged}{
  \condition{
        (TROLLEY > 0 && NURSES > 0) && (tagged_location == 1)
  }
  \action{
        next->TROLLEY = TROLLEY - 1;
        next->NURSES = NURSES - 1;
        next->AMBULANCE_PATIENT_ASSESSED =
              AMBULANCE_PATIENT_ASSESSED + 1;
        next->tagged_location=10;
  }
  \rate{
        TROLLEY > 1
        ?
        (rate_see_emergency_nurse * ((double)(1)) * (((double)(1)) /
        ((double)TROLLEY)))
        :
        rate_see_emergency_nurse
  }
}

%% complete assessment %%

\transition{T4}{
  \condition{
        (PATIENT_ASSESSED > 0 && tagged_location != 8) ||
        (PATIENT_ASSESSED > 1 && tagged_location == 8)
  }
  \action{
        next->PATIENT_ASSESSED = PATIENT_ASSESSED - 1;
        next->NURSES = NURSES + 1;
        next->ASSESSED_PATIENTS = ASSESSED_PATIENTS + 1;
  }
  \rate{
        (tagged_location == 8)
        ?
        (rate_complete_assessment * ((double)(PATIENT_ASSESSED - 1)) *
        (((double)(PATIENT_ASSESSED - 1)) /
        ((double)PATIENT_ASSESSED)))
        :
        (rate_complete_assessment * ((double)(PATIENT_ASSESSED)))
  }
}

\transition{T4_tagged}{
  \condition{
        (PATIENT_ASSESSED > 0) && (tagged_location==8)
```

```
    }
    \action{
          next->PATIENT_ASSESSED = PATIENT_ASSESSED - 1;
          next->NURSES = NURSES + 1;
          next->ASSESSED_PATIENTS = ASSESSED_PATIENTS + 1;
          next->tagged_location=11;
    }
    \rate{
          PATIENT_ASSESSED > 1
          ?
          (rate_complete_assessment * ((double)(1)) * (((double)(1)) /
          ((double)PATIENT_ASSESSED)))
          :
          rate_complete_assessment
    }
  }

  %% complete emergency assessment %%

  \transition{T5}{
    \condition{
          (AMBULANCE_PATIENT_ASSESSED > 0 && tagged_location != 10) ||
          (AMBULANCE_PATIENT_ASSESSED > 1 && tagged_location == 10)
    }
    \action{
          next->AMBULANCE_PATIENT_ASSESSED =
                AMBULANCE_PATIENT_ASSESSED - 1;
          next->NURSES = NURSES + 1;
          next->ASSESSED_PATIENTS = ASSESSED_PATIENTS + 1;
    }
    \rate{
          (tagged_location == 10)
          ?
          (rate_complete_emergency_assessment *
          ((double)(AMBULANCE_PATIENT_ASSESSED - 1)) *
          (((double)(AMBULANCE_PATIENT_ASSESSED - 1)) /
          ((double)AMBULANCE_PATIENT_ASSESSED)))
          :
          (rate_complete_emergency_assessment *
          ((double)(AMBULANCE_PATIENT_ASSESSED)))
    }
  }

  \transition{T5_tagged}{
    \condition{
          (AMBULANCE_PATIENT_ASSESSED > 0) && (tagged_location==10)
    }
    \action{
          next->AMBULANCE_PATIENT_ASSESSED =
                AMBULANCE_PATIENT_ASSESSED - 1;
          next->NURSES = NURSES + 1;
          next->ASSESSED_PATIENTS = ASSESSED_PATIENTS + 1;
          next->tagged_location=11;
    }
    \rate{
          AMBULANCE_PATIENT_ASSESSED > 1
```

```
        ?
        (rate_complete_emergency_assessment * ((double)(1)) *
        (((double)(1)) / ((double)AMBULANCE_PATIENT_ASSESSED)))
        :
        rate_complete_emergency_assessment
  }
}

%% surgery %%

\transition{T6}{
  \condition{
        (DOCTORS > 0 && WAITING_FOR_SURGERY > 0 &&
        tagged_location != 17) ||
        (DOCTORS > 0 && WAITING_FOR_SURGERY > 1 &&
        tagged_location == 17)
  }
  \action{
        next->DOCTORS = DOCTORS - 1;
        next->WAITING_FOR_SURGERY = WAITING_FOR_SURGERY - 1;
        next->SURGERY_DONE = SURGERY_DONE + 1;
  }
  \rate{
        (tagged_location == 17)
        ?
        ((WAITING_FOR_SURGERY < DOCTORS) ?
        (rate_surgery * ((double)(WAITING_FOR_SURGERY - 1)) *
        (((double)(WAITING_FOR_SURGERY - 1)) /
        ((double)WAITING_FOR_SURGERY))) :
        (rate_surgery * ((double)(DOCTORS)) *
        (((double)(WAITING_FOR_SURGERY - 1)) /
        ((double)WAITING_FOR_SURGERY))))
        :
        ((WAITING_FOR_SURGERY < DOCTORS) ? (rate_surgery *
        ((double)(WAITING_FOR_SURGERY))) :
        (rate_surgery * ((double)DOCTORS)))
  }
}

\transition{T6_tagged}{
  \condition{
        (DOCTORS > 0 && WAITING_FOR_SURGERY > 0) &&
        (tagged_location==17)
  }
  \action{
        next->DOCTORS = DOCTORS - 1;
        next->WAITING_FOR_SURGERY = WAITING_FOR_SURGERY - 1;
        next->SURGERY_DONE = SURGERY_DONE + 1;
        next->tagged_location=15;
  }
  \rate{
        WAITING_FOR_SURGERY > 1
        ?
        (rate_surgery * ((double)(1)) * (((double)(1)) /
        ((double)WAITING_FOR_SURGERY)))
        :
```

```
        rate_surgery
  }
}

%% see doctor %%

\transition{T7}{
  \condition{
        (DOCTORS > 0 && WAITING_FOR_DOCTOR > 0 &&
        tagged_location != 16) ||
        (DOCTORS > 0 && WAITING_FOR_DOCTOR > 1 &&
        tagged_location == 16)
  }
  \action{
        next->DOCTORS = DOCTORS - 1;
        next->WAITING_FOR_DOCTOR = WAITING_FOR_DOCTOR - 1;
        next->TREATED_BY_DOCTOR = TREATED_BY_DOCTOR + 1;
  }
  \rate{
        (tagged_location == 16)
        ?
        ((WAITING_FOR_DOCTOR < DOCTORS) ?
        (rate_see_doctor * ((double)(WAITING_FOR_DOCTOR - 1)) *
        (((double)(WAITING_FOR_DOCTOR - 1)) /
        ((double)WAITING_FOR_DOCTOR))) :
        (rate_see_doctor * ((double)(DOCTORS)) *
        (((double)(WAITING_FOR_DOCTOR - 1)) /
        ((double)WAITING_FOR_DOCTOR))))
        :
        ((WAITING_FOR_DOCTOR < DOCTORS) ? (rate_see_doctor *
        ((double)(WAITING_FOR_DOCTOR))) :
        (rate_see_doctor * ((double)DOCTORS)))
  }
}

\transition{T7_tagged}{
  \condition{
        (DOCTORS > 0 && WAITING_FOR_DOCTOR > 0) &&
        (tagged_location==16)
  }
  \action{
        next->DOCTORS = DOCTORS - 1;
        next->WAITING_FOR_DOCTOR = WAITING_FOR_DOCTOR - 1;
        next->TREATED_BY_DOCTOR = TREATED_BY_DOCTOR + 1;
        next->tagged_location=13;
  }
  \rate{
        WAITING_FOR_DOCTOR > 1
        ?
        (rate_see_doctor * ((double)(1)) * (((double)(1)) /
        ((double)WAITING_FOR_DOCTOR)))
        :
        rate_see_doctor
  }
}
```

```
%% perform lab tests %%

\transition{T8}{
  \condition{
        (WAITING_FOR_TESTS > 0 && tagged_location != 18) ||
        (WAITING_FOR_TESTS > 1 && tagged_location == 18)
  }
  \action{
        next->WAITING_FOR_TESTS = WAITING_FOR_TESTS - 1;
        next->TEST_DONE = TEST_DONE + 1;
  }
  \rate{
        (tagged_location == 18)
        ?
        (rate_perform_lab_tests *
        ((double)(WAITING_FOR_TESTS - 1)) *
        (((double)(WAITING_FOR_TESTS - 1)) /
        ((double)WAITING_FOR_TESTS)))
        :
        (rate_perform_lab_tests *
        ((double)(WAITING_FOR_TESTS)))
  }
}

\transition{T8_tagged}{
  \condition{
        (WAITING_FOR_TESTS > 0) && (tagged_location==18)
  }
  \action{
        next->WAITING_FOR_TESTS = WAITING_FOR_TESTS - 1;
        next->TEST_DONE = TEST_DONE + 1;
        next->tagged_location=12;
  }
  \rate{
        WAITING_FOR_TESTS > 1
        ?
        (rate_perform_lab_tests * ((double)(1)) *
        (((double)(1)) / ((double)WAITING_FOR_TESTS)))
        :
        rate_perform_lab_tests
  }
}

%% discharge treated patient %%

\transition{T9}{
  \condition{
        (TREATED_BY_DOCTOR > 0 && tagged_location != 13) ||
        (TREATED_BY_DOCTOR > 1 && tagged_location == 13)
  }
  \action{
        next->TREATED_BY_DOCTOR = TREATED_BY_DOCTOR - 1;
        next->HEALTHY = HEALTHY + 1;
        next->DOCTORS = DOCTORS + 1;
  }
```

```
    \rate{
          (tagged_location == 13)
          ?
          (rate_discharge_treated_patient *
          ((double)(TREATED_BY_DOCTOR - 1)) *
          (((double)(TREATED_BY_DOCTOR - 1)) /
          ((double)TREATED_BY_DOCTOR)))
          :
          (rate_discharge_treated_patient *
          ((double)(TREATED_BY_DOCTOR)))
    }
  }

  \transition{T9_tagged}{
    \condition{
          (TREATED_BY_DOCTOR > 0) && (tagged_location==13)
    }
    \action{
          next->TREATED_BY_DOCTOR = TREATED_BY_DOCTOR - 1;
          next->HEALTHY = HEALTHY + 1;
          next->DOCTORS = DOCTORS + 1;
          next->tagged_location=3;
    }
    \rate{
          TREATED_BY_DOCTOR > 1
          ?
          (rate_discharge_treated_patient * ((double)(1)) *
          (((double)(1)) / ((double)TREATED_BY_DOCTOR)))
          :
          rate_discharge_treated_patient
    }
  }

  %% to doctor %%

  \transition{T16}{
    \condition{
          (ASSESSED_PATIENTS > 0 && tagged_location != 11) ||
          (ASSESSED_PATIENTS > 1 && tagged_location == 11)
    }
    \action{
          next->ASSESSED_PATIENTS = ASSESSED_PATIENTS - 1;
          next->WAITING_FOR_DOCTOR = WAITING_FOR_DOCTOR + 1;
    }
    \weight{
          3.0
    }
  }

  \transition{T16_tagged}{
    \condition{
          (ASSESSED_PATIENTS > 0) && (tagged_location==11)
    }
    \action{
          next->ASSESSED_PATIENTS = ASSESSED_PATIENTS - 1;
          next->WAITING_FOR_DOCTOR = WAITING_FOR_DOCTOR + 1;
```

```
        next->tagged_location=16;
  }
  \weight{
        3.0
  }
}

%% to surgery %%

\transition{T17}{
  \condition{
        (ASSESSED_PATIENTS > 0 && tagged_location != 11) ||
        (ASSESSED_PATIENTS > 1 && tagged_location == 11)
  }
  \action{
        next->ASSESSED_PATIENTS = ASSESSED_PATIENTS - 1;
        next->WAITING_FOR_SURGERY = WAITING_FOR_SURGERY + 1;
  }
  \weight{
        1.0
  }
}

\transition{T17_tagged}{
  \condition{
        (ASSESSED_PATIENTS > 0) && (tagged_location==11)
  }
  \action{
        next->ASSESSED_PATIENTS = ASSESSED_PATIENTS - 1;
        next->WAITING_FOR_SURGERY = WAITING_FOR_SURGERY + 1;
        next->tagged_location=17;
  }
  \weight{
        1.0
  }
}

%% to tests %%

\transition{T18}{
  \condition{
        (ASSESSED_PATIENTS > 0 && tagged_location != 11) ||
        (ASSESSED_PATIENTS > 1 && tagged_location == 11)
  }
  \action{
        next->ASSESSED_PATIENTS = ASSESSED_PATIENTS - 1;
        next->WAITING_FOR_TESTS = WAITING_FOR_TESTS + 1;
  }
  \weight{
        2.0
  }
}

\transition{T18_tagged}{
  \condition{
        (ASSESSED_PATIENTS > 0) && (tagged_location==11)
```

```
    }
    \action{
          next->ASSESSED_PATIENTS = ASSESSED_PATIENTS - 1;
          next->WAITING_FOR_TESTS = WAITING_FOR_TESTS + 1;
          next->tagged_location=18;
    }
    \weight{
          2.0
    }
  }
}
```

A.3.2 PEPA Model

The PEPA model that corresponds to the GSPN model of Figure 6.22 is given below:

```
Healthy = (fall_ill, r1).Ill

Ill = (walk_in_arrival, r2).Waiting_Room +
      (ambulance_arrival, r3).Trolley

Waiting_Room = (see_nurse, r4).Patient_Being_Assessed

Patient_Being_Assessed = (complete_assessment, r5).Waiting_To_Be_Treated

Trolley = (see_emergency_nurse, r6).Ambulance_Patient_Being_Assessed

Ambulance_Patient_Being_Assessed = (complete_emergency_assessment, r7).
                                    Waiting_To_Be_Treated

Waiting_To_Be_Treated = (see_doctor, r8).Treated_By_Doctor +
                        (surgery, r9).Surgery_Done +
                        (perform_lab_tests, r10).Tests_Done

Treated_By_Doctor = (discharge_treated_patient, r11).Healthy

Surgery_Done = (recover, r12).Patient_Recovered

Patient_Recovered = (discharge_recovered_patient, r13).Healthy

Tests_Done = (evaluate_results, r14).Waiting_To_Be_Treated

Nurse = (see_nurse, r4).(complete_assessment, r5).Nurse +
        (see_emergency_nurse, r6).(complete_emergency_assessment, r7).
        Nurse

Doctor = (see_doctor, r8).(discharge_treated_patient, r11).Doctor +
         (surgery, r9).(recover, r12).Doctor

Patients = Healthy[PP]
```

```
Nurses = Nurse[NN]

Doctors = Doctor[DD]

AE_Unit = Patients <see_nurse, complete_assessment,
          see_emergency_nurse, complete_emergency_assessment,
          see_doctor, discharge_treated_patient,  surgery, recover>
          (Nurses <> Doctors))
```

Bibliography

[Agerwala79] T. Agerwala. “Putting Petri nets to work”. In *IEEE Computer*, pp. 85–94, December 1979.

[Ajmone Marsan84] M. Ajmone Marsan, G. Conte and G. Balbo. “A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems”. In *ACM Transactions on Computer Systems*, vol. 2, no. 2:93–122, May 1984.

[Ajmone Marsan95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Series in Parallel Computing. John Wiley & Sons, 1995. ISBN 0-471-93059-8.

[Alur91] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. Ph.D. thesis, Stanford University, 1991.

[Argent-Katwala06] A. Argent-Katwala. *A Compositional, Collaborative Performance Pipeline*. Ph.D. thesis, Imperial College, London, United Kingdom, November 2006.

[Argent-Katwala07a] A. Argent-Katwala and J. T. Bradley. “PerformDB: Community-driven Performance Modelling and Analysis”. In *UKPEW’07, Proceedings of the 23rd Annual UK Performance Engineering Workshop*, 2007.

[Argent-Katwala07b] A. Argent-Katwala, J. T. Bradley, A. Clark and S. Gilmore. “Location-aware Quality of Service Measurements for Service

Level Agreements”. In *TGC’07, Proceedings of the 3rd International Conference on Trustworthy Global Computing*, vol. 4912 of *LNCS*, pp. 222–239, 2007.

[Au-Yeung04] S. W. M. Au-Yeung, N. J. Dingle and W. J. Knottenbelt. “Efficient Approximation of Response Time Densities and Quantiles in Stochastic Models”. In *WOSP’04, Proceedings of the 4th International Workshop on Software and Performance,* pp. 151–155. ACM, Redwood City, January 2004.

[Aziz96] A. Aziz, K. Sanwal, V. Singhal and R. Brayton. “Verifying continuous-time Markov chains”. In *Computer-Aided Verification*, vol. 1102 of *LNCS*, pp. 269–276, 1996.

[Aziz00] A. Aziz, K. Sanwal, V. Singhal and R. Brayton. “Model checking continuous-time Markov chains”. In *ACM Transactions on Computational Logic*, vol. 1, no. 1:162–170, 2000.

[Baier00] C. Baier, B. R. Haverkort, H. Hermanns and J.-P. Katoen. “On the Logical Characterisations of Performability Properties”. In *ICALP’00, Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, vol. 1853 of *LNCS*, pp. 780–792, 2000.

[Baier03] C. Baier, B. R. Haverkort, H. Hermanns and J.-P. Katoen. “Model-checking algorithms for continuous-time Markov chains”. In *IEEE Transactions on Software Engineering,* vol. 29, no. 6:524–541, June 2003.

[Baier04] C. Baier, L. Cloth, B. R. Haverkort, M. Kuntz and M. Siegle. “Model Checking Action- and State-Labelled Markov Chains”. In *DSN’04, Proceedings of the 34th International Conference on Dependable Systems and Networks*, pp. 701–710, June 2004.

[Bause02] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg Verlag, Wiesbaden, Germany, 2002. 2nd edition.

[Bolch98] G. Bolch, S. Greiner, H. de Meer and K. S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, August 1998.

[Bonet07] P. Bonet, C. Lladó, R. Puijaner and W. J. Knottenbelt. "PIPE v2.5: A Petri Net Tool for Performance Modelling". In *CLEI'07, Proceedings of the 23rd Latin American Conference on Informatics*, 2007.

[Bradley03a] J. T. Bradley, N. J. Dingle, S. T. Gilmore and W. J. Knottenbelt. "Extracting Passage Times from PEPA models with the HYDRA Tool: a Case Study". In S. A. Jarvis (ed.), *UKPEW'03, Proceedings of 19th Annual UK Performance Engineering Workshop*, pp. 79–90. University of Warwick, July 2003.

[Bradley03b] J. T. Bradley, N. J. Dingle, P. G. Harrison and W. J. Knottenbelt. "Distributed Computation of Passage Time Quantiles and Transient State Distributions in Large Semi-Markov Models". In *PMEO-PDS'03, Proceedings of the International Workshop on Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems*, p. 281. IEEE Computer Society Press, Nice, April 2003.

[Bradley03c] J. T. Bradley, N. J. Dingle, W. J. Knottenbelt and P. G. Harrison. "Performance Queries on Semi-Markov Stochastic Petri Nets with an Extended Continuous Stochastic Logic". In *PNPM'03, Proceedings of the 10th International Workshop on Petri Nets and Performance Models*, pp. 62–71. University of Illinois at Urbana-Champaign, September 2003.

[Bradley03d] J. T. Bradley, N. J. Dingle, W. J. Knottenbelt and H. J. Wilson. "Hypergraph-based Parallel Computation of Passage Time Densities in Large Semi-Markov Models". In A. N. Langville and W. J. Stewart (eds.), *NSMC'03, Proceedings of the 4th International Workshop on Numerical Solutions of Markov Chains*, pp. 99–120. University of Illinois at Urbana-Champaign, September 2003.

[Bradley04] J. T. Bradley and W. J. Knottenbelt. "The ipc/HYDRA Tool Chain for the Analysis of PEPA Models". In B. Haverkort et al. (ed.), *QEST'04, Proceedings of the 1st IEEE Conference on the Quantitative Evaluation of Systems*, pp. 334–335. IEEE Computer Society Press, University of Twente, Enschede, September 2004.

[Bradley06] J. T. Bradley, N. J. Dingle, U. Harder, P. G. Harrison and W. J. Knottenbelt. "Response Time Densities and Quantiles in Large Markov and Semi-Markov Models". In *Performance Evaluation of Parallel, Distributed and Emergent Systems*, vol. 1. Nova Science Publishers, 2006.

[Bradley08] J. T. Bradley, R. A. Hayden, W. J. Knottenbelt and T. Suto. "Extracting Response Times from Fluid Analysis of Performance Models". In *SIPEW'08, Proceedings of the SPEC International Performance Evaluation Workshop*, vol. 5119 of *Lecture Notes in Computer Science*, pp. 29–43. Springer-Verlag, Darmstadt, Germany, June 2008.

[Brien08a] D. K. Brien. *Performance Trees: Implementation and Distributed Evaluation*. M.Sc. thesis, Imperial College London, June 2008.

[Brien08b] D. K. Brien, N. J. Dingle, W. J. Knottenbelt, H. Kulatunga and T. Suto. "Performance Trees: Implementation and Distributed Evaluation". In *PDMC'08, Proceedings of the 7th International Workshop on Parallel and Distributed Methods in Verification*, pp. 67–82. Elsevier, Budapest, Hungary, March 2008.

[Chiola95] G. Chiola, G. Franceschinis, R. Gaeta and M. Ribaudo. “GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets”. In *Performance Evaluation – Special Issue on Performance Modelling Tools*, vol. 24, no. 1–2:47–68, November 1995.

[Ciardo94] G. Ciardo, R. German and C. Lindemann. “A Characterization of the Stochastic Process Underlying a Stochastic Petri Net”. In *IEEE Transactions on Software Engineering*, vol. 20, no. 7:506–515, July 1994.

[Clark01] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders and P. Webster. “The Möbius Modeling Tool”. In *PNPM’01, Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pp. 241–250, September 2001.

[Clark07] A. Clark, S. Gilmore, J. Hillston and M. Tribastone. “Stochastic Process Algebras”. In *Formal Methods for Performance Evaluation*, LNCS Tutorials, pp. 132–179. Springer-Verlag, May 2007.

[Clarke Jr.01] E. M. Clarke Jr., O. Grumberg and D. A. Peled. *Model Checking*. MIT Press, 2001. ISBN 0-262-03270-8.

[Cooper81] R. B. Cooper. *Introduction to Queueing Theory*. Elsevier North Holland, 2nd ed., 1981. ISBN 0-444-00379-7.

[D’Aprile04] D. D’Aprile, S. Donatelli and J. Sproston. “CSL Model Checking for the GreatSPN Tool”. In *ISCIS’04, Proceedings of the 19th International Symposium on Computer and Information Sciences*, vol. 3280 of *Lecture Notes In Computer Science*, pp. 543–552. Springer-Verlag, 2004.

[Dingle03] N. J. Dingle, W. J. Knottenbelt and P. G. Harrison. “HYDRA: HYpergraph-based Distributed Response-time Analyser”. In H. R. Arabnia and Y. Man (eds.), *PDPTA’03, Proceedings of the 2003*

International Conference on Parallel and Distributed Processing Techniques and Applications, vol. 1, pp. 215–219. Las Vegas, NV, June 2003.

[Dingle04a] N. J. Dingle. *Parallel Computation of Response Time Densities and Quantiles in Large Markov and Semi-Markov Models*. Ph.D. thesis, Imperial College London, October 2004.

[Dingle04b] N. J. Dingle, P. G. Harrison and W. J. Knottenbelt. “Uniformization and Hypergraph Partitioning for the Distributed Computation of Response Time Densities in Very Large Markov Models”. In *Journal of Parallel and Distributed Computing*, vol. 64, no. 8:908–920, August 2004.

[Dingle08a] N. J. Dingle and W. J. Knottenbelt. “Automated Customer-Centric Performance Analysis of Generalised Stochastic Petri Nets using Tagged Tokens”. In *PASM’08, Proceedings of the 3rd International Workshop on Practical Applications of Stochastic Modelling*. Palma de Mallorca, Mallorca, Spain, September 2008.

[Dingle08b] N. J. Dingle, W. J. Knottenbelt, H. Kulatunga and T. Suto. “A Parallel and Distributed Analysis Pipeline for Performance Tree Evaluation”. In *QEST’08, Proceedings of the 5th International Conference on the Quantitative Evaluation of Systems*. IEEE Computer Society, Saint Malo, France, September 2008.

[Donatelli95] S. Donatelli, M. Ribaudo and J. Hillston. “A comparison of Performance Evaluation Process Algebra and generalized stochastic Petri nets”. In *PNPM’95, Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, pp. 158 – 169. IEEE Computer Society, 1995.

[Grassman87] W. Grassman. “Means and Variances of Time Averages in Markovian Environments”. In *European Journal of Operational Research*, vol. 31, no. 1:132–139, 1987.

[Grunske08a] L. Grunske. “Specification Patterns for Probabilistic Quality Properties”. In *ICSE’08, Proceedings of the 30th International Conference on Software Engineering*, pp. 31–40. ACM, 2008.

[Grunske08b] L. Grunske, K. Winter and N. Yatapanage. “Defining the Abstract Syntax of Visual Languages with Advanced Graph Grammars – A Case Study based on Behavior Trees”. In *Journal of Visual Languages and Computing*, vol. 19, no. 3:343–379, 2008.

[Harrison02] P. G. Harrison and W. J. Knottenbelt. “Passage-time Distributions in Large Markov Chains”. In M. Martonosi and E. A. de Souza e Silva (eds.), *Proceedings of ACM SIGMETRICS 2002*, pp. 77–85, Marina Del Rey, USA, June 2002.

[Hermanns00] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser and M. Siegle. “Towards Model Checking Stochastic Process Algebra”. In *IFM’00, Proceedings of the 2nd International Conference on Integrated Formal Methods*, pp. 420–439, November 2000.

[Hillston94] J. Hillston. *A Compositional Approach to Performance Modelling*. Ph.D. thesis, Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, United Kingdom, 1994. CST–107–94.

[Hillston04] J. Hillston. “Modelling and Simulation”. *Lecture notes*, University of Edinburgh, 2004.

[Hillston05] J. Hillston. “Fluid Flow Approximation of PEPA models”. In *QEST’05, Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems*. IEEE Computer Society Press, Italy, September 2005.

[Hirel00] C. Hirel, R. Sahner, X. Zhang and K. S. Trivedi. “Reliability and Performability Modeling Using SHARPE 2000”. In *TOOLS’00, Proceedings of the 11th International Conference on Computer*

Performance Evaluation, Modelling Techniques and Tools, vol. 1786 of *LNCS*, p. 345, 2000.

[Howard71] R. A. Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*, vol. 2 of *Series in Decision and Control*. John Wiley & Sons, 1971.

[Jansen05] D. N. Jansen and H. Hermanns. “QoS Modelling and Analysis with UML-Statecharts: the StoCharts Approach”. In *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 4:28–33, March 2005.

[Jensen53] A. Jensen. “Markoff Chains as an Aid in the Study of Markoff Processes”. In *Skandinavian Aktuarietidskr.*, vol. 36:87–91, 1953.

[Katoen01] J.-P. Katoen, M. Kwiatkowska, G. Norman and D. Parker. “Faster and Symbolic CTMC Model Checking”. In L. de Alfaro and S. Gilmore (eds.), *Proceedings of Process Algebra and Probabilistic Methods*, vol. 2165 of *Lecture Notes in Computer Science*, pp. 23–38. Springer-Verlag, Aachen, September 2001.

[Kendall53] D. G. Kendall. “Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Embedded Markov chain”. In *The Annals of Mathematical Statistics*, vol. 24, no. 3:338–354, September 1953.

[Knottenbelt96] W. J. Knottenbelt. *Generalised Markovian Analysis of Timed Transitions Systems*. M.Sc. thesis, University of Cape Town, South Africa, July 1996.

[Knottenbelt09] W. J. Knottenbelt, N. J. Dingle and T. Suto. “Parallel and Distributed Evaluation of Performance Tree Queries”. In *PARENG’09, Proceedings of the 1st International Conference on Parallel, Distributed and Grid Computing for Engineering*. Pécs, Hungary, Book Chapter Accompanying Invited Lecture, 32 pages, To Appear in April 2009.

[Kwiatkowska02] M. Kwiatkowska, G. Norman and D. Parker. “PRISM: Probabilistic Symbolic Model Checker”. In A. J. Field et al. (ed.), *TOOLS’02, Proceedings of the 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, vol. 2324 of *Lecture Notes in Computer Science*, pp. 200–204. Springer-Verlag, London, 2002.

[Lee94] I. Lee, P. Bremond-Gregoire and R. Gerber. “A Process Algebraic Approach to the Specification and Analysis of Resource-bound Real-Time Systems”. In *Proceedings of the IEEE*, vol. 82, no. 1, January 1994.

[Lee97] I. Lee and O. Sokolsky. “A Graphical Property Specification Language”. In *HASE’97, Proceedings of the 2nd High-Assurance Systems Engineering Workshop*, pp. 42–47. IEEE Computer Society, Washington, DC, USA, November 1997.

[López-Grao04] J. P. López-Grao, J. Merseguer and J. Campos. “From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering”. In *WOSP’04, Proceedings of the 4th International Workshop on Software and Performance*, pp. 243–244. Redwood City, CA, January 2004.

[Melamed84] B. Melamed and M. Yadin. “Randomization Procedures in the Computation of Cumulative-Time Distributions Over Discrete State Markov Processes”. In *Operations Research*, vol. 32, no. 4:926–944, July–August 1984.

[MeSC] MeSC. “The Midlands e-Science Grid Cluster”. `http://www.ep.ph.bham.ac.uk/cluster/`.

[Meyer80] J. F. Meyer. “On Evaluating the Performability of Degradable Computing Systems”. In *IEEE Transactions on Computers*, vol. C-29, no. 8:720–731, August 1980.

[Miner03] A. S. Miner. “Computing Response Time Distributions using Stochastic Petri Nets and Matrix Diagrams”. In *PNPM’03, Proceedings of the 10th International Workshop on Petri nets and Performance Models*, pp. 10–19. Urbana-Champaign, IL, September 2003.

[Mitrani98] I. Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998. ISBN 0-521-58511-2.

[Molloy81] M. K. Molloy. *On the Integration of Delay and Throughout Measures in Distributed Processing Models*. Ph.D. thesis, University of California, 1981.

[Molloy82] M. K. Molloy. “Performance Analysis using Stochastic Petri Nets”. In *IEEE Transactions on Computers*, vol. 31, no. 9:913–917, September 1982.

[Muppala92] J. K. Muppala and K. S. Trivedi. “Numerical Transient Analysis of Finite Markovian Queueing Systems”. In U. N. Bhat and I. V. Basawa (eds.), *Queueing and Related Models*, pp. 262–284. Oxford University Press, 1992.

[Murata89] T. Murata. “Petri Nets: Properties, Analysis and Applications”. In *Proceedings of the IEEE*, vol. 77, no. 4:541–580, April 1989.

[Natkin81] S. Natkin. *Les Reseaux de Petri Stochastiques et leur Application a l’Évaluation des Systèmes Informatiques*. Ph.D. thesis, University of California, 1981.

[Nelson95] R. Nelson. *Probability, Stochastic Processes and Queueing Theory: The Mathematics of Computer Performance Modeling*. Springer-Verlag, 1995. ISBN 0-387-94452-4.

[OMG07] OMG. “Unified Modeling Language Specification, v. 2.1.2”. `http://www.omg.org/spec/UML/2.1.2/`, November 2007.

[Peterson77] J. L. Peterson. “Petri Nets”. In *ACM Computing Surveys*, vol. 9, no. 3:223–252, September 1977.

[Peterson81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[Petri62] C. A. Petri. *Kommunikation mit Automaten*. Ph.D. thesis, Universität Bonn, 1962.

[PIPE] PIPE. “*PIPE2*: Platform-Independent Petri net Editor”. `http://pipe2.sourceforge.net`.

[PNML] PNML. “The Petri Net Markup Language”. `http://www2.informatik.hu-berlin.de/top/pnml/`.

[Powell02] T. Powell. “Responsiveness vs. Performance”. `http://www.worldtimzone.com/blog/date/2002/09/11/responsiveness-vs-performance/`, 2002.

[Pyke61a] R. Pyke. “Markov Renewal Processes: Definitions and Preliminary Properties”. In *Annals of Mathematical Statistics*, vol. 32, no. 4:1231–1242, December 1961.

[Pyke61b] R. Pyke. “Markov Renewal Processes with Finitely Many States”. In *Annals of Mathematical Statistics*, vol. 32, no. 4:1243–1259, December 1961.

[Reibman88] A. Reibman and K. S. Trivedi. “Numerical Transient Analysis of Markov Models”. In *Computers and Operations Research*, vol. 15, no. 1:19–36, 1988.

[Reisig85] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[Ross82] S. M. Ross. *Stochastic processes*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, 1982. ISBN 0-471-12062-6.

[Suto05] T. Suto. "GRAIL: Grid-Enabled Performance Analysis Using Stochastic Logics". In *PASTA'05, Proceedings of the 4th Workshop on Process Algebra and Stochastically Timed Activities*. University of Edinburgh, Edinburgh, Scotland, United Kingdom, September 2005.

[Suto06a] T. Suto. "Expanding the Boundaries of Performance Requirement Representation with Performance Trees". In *PASTA'06, Proceedings of the 5th Workshop on Process Algebra and Stochastically Timed Activities*, pp. 108–116. Imperial College London, London, England, United Kingdom, June 2006.

[Suto06b] T. Suto, J. T. Bradley and W. J. Knottenbelt. "Performance Trees: A New Approach to Quantitative Performance Specification". In *MASCOTS'06, Proceedings of the 14th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 303–313. IEEE Computer Society, Monterey, California, USA, September 2006.

[Suto07] T. Suto, J. T. Bradley and W. J. Knottenbelt. "Performance Trees: Expressiveness And Quantitative Semantics". In *QEST'07, Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems*, pp. 41–50. IEEE Computer Society, Edinburgh, Scotland, United Kingdom, September 2007.

[Suto08a] T. Suto, J. T. Bradley, D. K. Brien, N. J. Dingle, W. J. Knottenbelt and H. Kulatunga. "Performance Trees: Application in a Distributed Analysis Environment". In *IEEE Transactions on Software Engineering*, 2008. Submitted for review.

[Suto08b] T. Suto and W. J. Knottenbelt. "*PIPE2*: A Tool for Parallel and Distributed Performance Evaluation". In *ACM SIGMETRICS Performance Evaluation Review*, vol. Special Issue, 2008. Invited paper. To appear.

[Tribastone07] M. Tribastone. "The PEPA Plug-in Project". In *QEST'07, Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems*, pp. 53–54. IEEE Computer Society, Edinburgh, Scotland, United Kingdom, September 2007.

[Trifunović04] A. Trifunović. *Parallel Algorithms for Hypergraph Partitioning*. Ph.D. thesis, Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, United Kingdom, 2004.

[Trivedi02] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, 2002.

[Wang08] L. Wang, N. J. Dingle and W. J. Knottenbelt. "Natural Language Specification of Performance Trees". In *EPEW'08, Proceedings of the 5th European Performance Engineering Workshop*, 2008.

[Younes05] H. L. S. Younes. "Ymer: A Statistical Model Checker". In K. Etessami and S. Rajamani (eds.), *Proceedings of the 17th International Conference on Computer Aided Verification*, vol. 3576 of *Lecture Notes in Computer Science*, pp. 429–433. Springer-Verlag, Edinburgh, Scotland, United Kingdom, 2005.

[Zhang05] Y. Zhang, D. Parker and M. Kwiatkowska. "Grid-enabled Probabilistic Model Checking with PRISM". In *AHM'05, Proceedings of the 4th All Hands Meeting*. Nottingham, United Kingdom, September 2005.