Performance Trees: A New Approach to Quantitative Performance Specification

Tamas Suto
Department of Computing
Imperial College London
180 Queen's Gate
SW7 2BZ London
suto@doc.ic.ac.uk

Jeremy T. Bradley
Department of Computing
Imperial College London
180 Queen's Gate
SW7 2BZ London
jb@doc.ic.ac.uk

William J. Knottenbelt
Department of Computing
Imperial College London
180 Queen's Gate
SW7 2BZ London
wjk@doc.ic.ac.uk

Abstract

We introduce Performance Trees (PTs), a novel representation formalism for the specification of model-based performance queries. Traditionally, stochastic logics have been the prevalent means of performance requirement expression; however, in practice, their use amongst system designers is limited on account of their inherent complexity and restricted expressive power. PTs are a more accessible alternative, in which performance queries are represented by hierarchical tree structures. This allows for the convenient visual composition of complex performance questions, and enables not only the verification of stochastic requirements, but also the direct extraction of performance measures. In addition, PTs offer a superset of the expressiveness of Continuous Stochastic Logic (CSL) since all CSL formulae can be translated into PT form.

Performance Trees can be used to represent passage time, transient, steady-state and higher order queries of varying levels of sophistication. While they are conceptually independent of the underlying stochastic modelling formalism, in many cases the tree operators we use are already backed up by good algorithmic and tool support for both stochastic verification and performance measure extraction. We do not therefore perceive major barriers to the integration of PTs into existing stochastic model checking tools. Indeed, we illustrate how semi-Markov passage time computation algorithms, based on numerical Laplace transform inversion, can be directly applied to the resolution of a case study PT query.

1 Introduction

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

a. construct a mathematical model of its operation (using some stochastic modelling formalism),

b. express associated performance-related queries in terms of requirements and measures (using a stochastic logic or other methodology),

c. apply specialised stochastic model checking or quantitative analysis software to resolve the queries.

This paper focuses primarily on step (b) of the process, but we also present a brief overview of steps (a) and (c) in order to convey a clearer understanding of the wider context of our work.

1.1 Stochastic Models

Many real-life systems exhibit random or probabilistic behaviour, which makes it difficult to predict individual events. However, it is often possible to use probability distributions to characterise and model this behaviour mathematically. Stochastic modelling formalisms encapsulate such systems of distributions in an elegant manner. Various formalisms exist, each of which characterises system behaviour at a different level of abstraction. We distinguish between two levels of models. *Low-level* modelling formalisms, such as Markov processes, semi-Markov processes (SMPs) [21] and generalised semi-Markov processes (GSMPs) [20], provide a raw representation of the system in terms of its states and transitions, and in many cases are amenable to numerical analysis. *High-level* models, such as stochastic Petri nets (SPNs) [1] and stochastic process algebras (SPAs) [12, 15] abstract the level of detail required for model specification and avoid the need for the tedious enumeration of every system state and transition. Whilst

it is sometimes the case that analysis can be performed on the high-level model directly, these are usually mapped onto low-level models for detailed performance analysis.

1.2 Performance Query Specification

Having created a stochastic model of the system, it needs to be decided what performance measures are of interest. For example, it may be part of a service level agreement that 95% of the time, an SMS message should take less than 5 seconds to travel between mobile phone handsets. It is common to capture such requirements in a logical formula, using a language such as CSL (see Section 2.1). This exploits the strength of logical performance specification, namely the ability to compose performance-related requirements systematically and concisely.

It might be natural to assume that the task of posing performance queries is solved by the use of stochastic logics, but it is not as simple as that. Two criticisms of such formalisms might be that they obscure the question being asked and do not provide the modeller with a complete set of usable performance questions. Many performance questions of value to system designers can not be asked, due to the limitation in expressiveness of current stochastic logics. Such questions may relate to performance measures that need to be extracted from the model directly in some way. In addition, performance engineers are often unsure as to what measures of performance should be used during analyses in order to obtain the most relevant feedback. Meaningful analysis results can only be expected if the system designer has a complete understanding of the full range of performance queries that can be expressed. In this paper, we seek to present a new framework for performance query specification which:

1. allows the specification of performance queries in a clearer and more accessible way,
2. provides an enlarged set of performance questions to the modeller and allows not only performance requirement verification, but also the extraction of quantitative performance measures of interest,
3. maintains the ability to express performance measures concisely, compositionally and systematically.

1.3 Stochastic Model Checking

Once the model representation of the system is available and the necessary performance-related queries have been specified, they can be submitted to a model checker (such as PRISM [19], ETMCC [14], MRMC [17] or the APNNtoolbox [9]) for validation, or to a quantitative analyser (such as SHARPE [16], DNAmaca [18], SPNP [10] or Möbius [11]) for measure extraction.

Most current model checkers are not enabled to perform performance measure extractions; therefore a significant limitation exists that constrains the scope of system analysis. We aim to address this problem when further developing tool support for our formalism, and intend to implement a combined model checker and quantitative analyser.

The remainder of this paper is organised as follows. In the next section, we present various flavours of stochastic logics for performance-related requirement representation. Section 3 provides an overview of various types of useful performance queries. Section 4 introduces the Performance Tree formalism for performance query representation, which is contrasted with stochastic logics in Section 5. Section 6 shows a worked example of a Performance Tree query with numerical results. Section 7 concludes and considers future work.

2 Logical Performance Specification Formalisms

This section introduces related performance specification formalisms. It also serves to give an overview of the distinct types of performance query that these different formalisms address. This catalogue of performance questions was the foundation of the design of the Performance Tree specification system of Section 4.

2.1 CSL

The most prevalent performance-enabled logic is *Continuous Stochastic Logic (CSL)* [2, 3, 5, 6], which can be considered to provide the framework for all other extended stochastic logic formalisms. CSL operates on continuous-time Markov chains on the state level. Performance requirements are expressed as formulae, which 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 power to express steady-state, path-based and nested constraints. The syntax for these constructs is as follows:

$$\begin{aligned} \sigma &\stackrel{\text{def}}{=} tt \mid a \mid \neg\sigma \mid \sigma \wedge \sigma \mid \mathcal{S}_{\bowtie p}(\sigma) \mid \mathcal{P}_{\bowtie p}(\varphi) \\ \varphi &\stackrel{\text{def}}{=} \mathcal{X}^I\sigma \mid \sigma\,\mathcal{U}^I\sigma \end{aligned}$$

tt represents a truth value, while atomic proposition a holds in state σ if σ is labelled with a. $\mathcal{S}_{\bowtie p}(\sigma)$ asserts that the aggregate steady-state probability for the states satisfying σ meets the bound $\bowtie\ p$, i.e. either $\leq\ p$ or $\geq\ p$, since $\bowtie\ \in \{\leq, \geq\}$. $\mathcal{P}_{\bowtie p}(\varphi)$ expresses a constraint on the paths satisfying φ to meet the bound $\bowtie\ p$. Paths are defined by φ, which can take the form of $\mathcal{X}^I\sigma$ or $\sigma_1\,\mathcal{U}^I\sigma_2$. The $\mathcal{X}^I\sigma$ path formula asserts that a transition is made to a σ state at

some time, $t \in I$, while $\sigma_1 \, \mathcal{U}^I \sigma_2$ asserts that σ_2 is satisfied at some time instant in the time interval I, while at all preceding time instants σ_1 holds.

2.2 aCSL / asCSL

aCSL [13] is an action-oriented variant of CSL, which is able to describe behaviours of interest. It is derived from stochastic process algebras, which model systems by describing their possible action behaviours. The fundamental premise underlying aCSL is that a state is more naturally described by the action behaviour it exhibits, and thus performance measures should be specified, in part, using action constraints. This logic enables the reasoning about system behaviours on a state-based model, enhanced with relevant action information. State formulae are defined just like the σ formulae in CSL, however, aCSL augments path formulae with the following:

$$\varphi \stackrel{\text{def}}{=} \sigma \; {}_A\mathcal{U}^{<t} \sigma \mid \sigma \; {}_A\mathcal{U}^{\leq t}_B \sigma$$

The path formula $\sigma_1 \; {}_A\mathcal{U}^{<t}\sigma_2$ is satisfied by a path if a σ_2 state is eventually reached in at most t time units by visiting σ_1 states and performing A-transitions only. $\sigma_1 \; {}_A\mathcal{U}^{\leq t}_B \sigma_2$ requires in addition that a move into a σ_2 state is performed and that this move is effected by transition B. The formula $\mathcal{X}^{<t}_A \sigma$ can be derived, since $\mathcal{X}^{<t}_A \sigma \equiv tt \; {}_\emptyset\mathcal{U}^{<t}_A \sigma$.

asCSL [4] was originally developed to characterise execution paths of action- and state-labelled Markov chains and to subsume CSL and aCSL. Path properties are characterised by regular expressions over actions and state formulae, in conjunction with time bounds. The CSL path formulae need to be amended in the following manner:

$$\varphi \stackrel{\text{def}}{=} \epsilon \mid \sigma \; b \mid \varphi \; ; \; \varphi \mid \varphi \cup \varphi \mid \varphi^*$$

where ϵ is the empty word and $b \in Act \cup Act'$, given that $Act' \notin Act$, where Act is a finite set of action labels and Act' is a pseudo-action which effects no state change. ';' denotes sequential composition, *i.e.* concatenation; $\cup$ represents alternative choice, and * stands for the n-fold sequential composition for arbitrary $n \geq 0$.

2.3 eCSL

eCSL [7] is defined over a higher level of models, namely semi-Markov stochastic Petri nets (SM-SPNs). These are semi-Markov extensions of conventional stochastic Petri nets whose state spaces map onto underlying semi-Markov processes. In contrast to other logical formalisms, eCSL operates on the model level, rather than at the state level. It was designed to represent a broader spectrum of performance requirements, including constraints on transient state distributions. eCSL does not support compound formulae, and thereby simplifies the representation mechanism. It also introduces separate layers for the specification of sets of states and of performance criteria. Its syntax is as follows:

$$\begin{aligned} \sigma &\stackrel{\text{def}}{=} tt \mid \neg\sigma \mid \sigma \wedge \sigma \mid p[N] \\ \varphi &\stackrel{\text{def}}{=} tt \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{S}_\rho(\sigma) \mid \mathcal{T}^\tau_\rho(\sigma, \, \sigma) \mid \\ &\quad \mathcal{P}^\tau_\rho(\sigma, \, \sigma) \end{aligned}$$

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

3 Performance Query Classification

In this section, we classify the types of performance queries that are useful to performance modellers, and present examples to demonstrate their application. In doing so, we will distinguish between two kinds of performance uery: a *performance requirement* and a *performance measure*. A performance requirement describes a property related to stochastic behaviour that must be satisfied by a system model; such a requirement is traditionally phrased in terms of a stochastic logic formula, which can be verified as true or false by a stochastic model checking tool. A performance measure is a more general concept that enables the extraction of quantitative measures of various kinds from a system model. These measures include densities or quantiles of response time, mean time to failure, system throughput and so on. Evaluation of performance queries is enabled by quantitative analysis tools.

3.1 Passage time queries

Passage time queries address requirements for the time needed for the system to reach a particular state or a set of states, having started from a given state or set of states. These requirements are useful when measuring system responsiveness. We can see from the examples below that even from a passage time query, distinct return types are possible, according to the performance quantity that is required:

1. *"What is the distribution of time for the passage between the set of states S and the set of states T?"*
2. *"What is the average time until the system reaches the set of states T, having started from the set of states S?"*

3. *"Does a passage between the set of states S and the set of states T occur within the time interval* $[0, 10]$ *with probability lying in the range* $[0.9, 0.98]$*?"*

Query 1 is expecting a (cumulative) distribution function for the passage time, Query 2 an average value (the first moment) of the passage time and Query 3 a truth value. When considering passages, we might want to narrow them even further by specifying that we require the passage to exclude a given set of states, meaning that the set of states should not be encountered en route to the target states. The following query illustrates these restrictions:

4. *"What is the probability of the passage from the set of states S to the set of states T completing in* 71 *time units, provided that the set of states E must be avoided along the passage?"*

3.2 Transient queries

Transient queries relate to the probability of the system being in a particular state or a set of states at a time instant t. They can be used to assess system reliability, since they can take the likelihood of the system entering a failure mode at a given time into account. Transient queries typically have three return types, namely a probability measure, a set of states or a truth value, *e.g.*

5. *"Is the probability that the system will be in the set of states T at time instant* 40*, given that the system has originally started from the set of states S, greater than* 0.87*?"*

6. *"What states does the system occupy at time instant* 40 *with probability exceeding* 0.2*?"*

3.3 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" (often referred to as the "steady-state"). Examples of different return types from steady-state queries include:

7. *"What is the steady-state probability of being in one of the set of states T?"*

8. *"Out of the set of states S, which states have a steady-state probability greater than* 0.12*?"*

Query 8 is interested in finding a probability value and Query 9 aims at obtaining a set of states as the result. A special kind of steady-state measure is the firing rate of a transition. It is classified into this category, because steady-state values are used in the relevant numerical calculations. Query 10 provides an example for this:

9. *"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 mean firing rate of action 'processed at B' multiplied by 200?"*

3.4 Higher order queries

Queries can be extended to include additional restrictions. A change of state in a system is effected by an action. Hence, it may be of interest to specify, as in the previous case, a set of included or a set of excluded actions, in order to observe the behaviour and development of the system, given that certain actions do or do not occur. The following examples illustrate this concept:

10. *"What is the variance of the passage time defined over the set of start states S and the set of target states T, with the constraint that action* $'processed'$ *takes place at least once and that actions* $'halt'$ *and* $'fail'$ *do not occur during the passage?"*

As mentioned previously, queries can consist of multiple performance requirements and measures. These can be composed together, with the result that the model checker or quantitative analyser will evaluate the individual requirements sequentially and produce separate results. Such queries could take the following form:

11. *"What is the probability that a passage from the set of states S to the set of states T will complete in* 50 *time units, and what is the density of time that it takes to complete this passage?"*

12. *"What is the average time required to complete the passage defined by the convolution of the passage from the set of start states S1 to the set of target states T1 with the passage from the set of start states S2 to the set of target states T2, having the additional constraint that the set of states E is excluded from both passages?"*

4 A Tree Formalism for Query Representation

In the past, as outlined in Section 2, various logical formalisms have been designed for the representation of performance requirements. However, even though these logical formalisms are very powerful, they lack the accessibility necessary to attract a larger and more diverse user base, and they are also unable to express certain types of performance

requirements. As can be seen from the previous examples, performance queries can become fairly complex. Hence, it is desirable to devise a concise, yet complete representation that summarises the essence of a query in a less esoteric way than stochastic logics do. To cater for this requirement and to overcome the indicated shortcomings, we have developed the *Performance Tree* formalism, a graphical alternative for compositional performance query specification, based on a hierarchical tree structure. We propose a general framework that allows for the expression of a wide range of performance requirements and measures in a uniform manner, regardless of the underlying modelling formalism.

A particular instance of a Performance Tree consists of various nodes interconnected by arcs. Nodes in the tree can be of two types. They can either represent operations or values. An *operation node* has inputs and outputs, which are represented by subnodes and supernodes respectively. They symbolise operations that will be performed during the verification and analysis procedure. *Value nodes*, on the other hand, only carry information that form the fundamental building blocks of the performance query, and hence have no subnodes. They symbolise the arguments of the operations. The following syntax describes the operation and the possible arguments (subnodes) of each node.

4.1 Syntax

4.1.1 Operation Nodes

The **?** operator is the topmost node of a tree. It represents the overall result of the performance query.

$$\begin{aligned} ? \quad ::= \quad & ; \mid \curlywedge \mid \neg \mid \bowtie \mid \oplus \mid \mathit{PTD} \mid \mathit{Dist} \mid \mathit{Moment} \\ & \mid \mathit{Conv} \mid \mathit{InInterval} \mid \mathit{InStates} \mid \mathit{ProbInInterval} \\ & \mid \mathit{ProbInStates} \mid \mathit{SS{:}P} \mid \mathit{SS{:}S} \mid \mathit{FR} \mid \mathit{StatesAtTime} \end{aligned}$$

The **;** operator is the sequential execution operator, which allows multiple performance requirements and measures to be composed together into one performance query. This operator is especially useful for the identification of optimisation opportunities across several sub-queries. The operator must have at least two subnodes and its result is the list of combined results of the individual sub-queries it combines.

$$\begin{aligned} ; \quad ::= \quad & (\curlywedge \mid \neg \mid \bowtie \mid \oplus \mid \mathit{PTD} \mid \mathit{Dist} \mid \mathit{Moment} \\ & \mid \mathit{Conv} \mid \mathit{InInterval} \mid \mathit{InStates} \mid \mathit{ProbInInterval} \\ & \mid \mathit{ProbInStates} \mid \mathit{SS{:}P} \mid \mathit{SS{:}S} \mid \mathit{FR} \\ & \mid \mathit{StatesAtTime})^{2..*} \end{aligned}$$

The $\curlywedge$ operator performs a boolean disjunction or conjunction operation. $\curlywedge \in \{\vee, \wedge\}$. It has two arguments, both of which need to represent a truth value.

$$\curlywedge \quad ::= \quad (\curlywedge \mid \neg \mid \bowtie \mid \mathit{InInterval} \mid \mathit{InStates} \mid \mathit{Bool})^2$$

The $\neg$ operator performs a boolean negation operation. It has one argument, which must represent a truth value.

$$\neg \quad ::= \quad \neg \mid \curlywedge \mid \bowtie \mid \mathit{InInterval} \mid \mathit{InStates} \mid \mathit{Bool}$$

The $\bowtie$ operator performs a binary comparison operation. $\bowtie \in \{<, \leq, =, \geq, >\}$. It has two arguments, both of which have a numerical value, and it returns a truth value.

$$\begin{aligned} \bowtie \quad ::= \quad & ([\oplus \mid \mathit{Num} \mid \mathit{Moment} \mid \mathit{ProbInInterval} \mid \\ & \mathit{ProbInStates} \mid \mathit{SS{:}P} \mid \mathit{FR}] \times (\oplus \mid \mathit{Num})) \\ & \mid \mathit{Moment, Moment} \mid \mathit{SS{:}P, SS{:}P} \mid \mathit{FR, FR} \\ & \mid \mathit{ProbInInterval, ProbInInterval} \\ & \mid \mathit{ProbInStates, ProbInStates} \end{aligned}$$

The $\oplus$ operator represents an arithmetic operation. $\oplus \in \{+, -, *, \div\}$. It has two arguments, both of which have a numerical value, and it returns a numerical result.

$$\begin{aligned} \oplus \quad ::= \quad & ([\oplus \mid \mathit{Num} \mid \mathit{Moment} \mid \mathit{ProbInInterval} \\ & \mid \mathit{ProbInStates} \mid \mathit{SS{:}P} \mid \mathit{FR}] \times (\oplus \mid \mathit{Num})) \\ & \mid \mathit{Moment, Moment} \mid \mathit{SS{:}P, SS{:}P} \mid \mathit{FR, FR} \\ & \mid \mathit{ProbInInterval, ProbInInterval} \\ & \mid \mathit{ProbInStates, ProbInStates} \end{aligned}$$

The **PTD** operator represents a passage time density. The arguments (subnodes) define the passage and the node itself represents the density of time for the passage to take place between two sets of states. There must always be at least two sets of states provided (start and target states), but optional constraints relating to actions, states or rewards (represented by the range operator $[\![\ldots]\!]$) can also be supplied.

$$\begin{aligned} \mathit{PTD} \quad ::= \quad & \mathit{States}^{2..4} \mid \mathit{States}^{2..4}, \mathit{Actions}^{1..2} \mid \mathit{States}^{2..4}, \\ & [\![\ldots]\!] \mid \mathit{States}^{2..4}, \mathit{Actions}^{1..2}, [\![\ldots]\!] \end{aligned}$$

The **Dist** operator represents a (cumulative) passage time distribution. It takes a passage time density as an argument and converts it into a passage time distribution.

$$\mathit{Dist} \quad ::= \quad \mathit{PTD}$$

The **Conv** operator represents the convolution of two passage time densities or distributions. The operator takes two arguments, which can both either be densities or distributions, and returns the convoluted function.

$$\mathit{Conv} \quad ::= \quad \mathit{PTD, PTD} \mid \mathit{Dist, Dist}$$

The **Moment** operator represents the raw moments of a

passage time density function. A single moment generating function can provide us with multiple valuable metrics, since we can derive any number of central moments, the first of which is the expected value, second the variance, *etc.* The operator has two arguments; the first being the rank of the moment that we want to calculate (*e.g.* 3^{rd} moment) and the second being the passage time density, distribution or convolution that we calculate the moments from. The result is a numerical value.

$$\textit{Moment} \quad ::= \quad \textit{Num, PTD} \mid \textit{Num, Dist} \mid \textit{Num, Conv}$$

The **SS:P** operator represents the steady-state probability for a given set of states.

$$\textit{SS:P} \quad ::= \quad \textit{States}$$

The **SS:S** operator represents the set of states that out of a specified set of states has a steady-state probability of a certain value or within a certain range.

$$\textit{SS:S} \quad ::= \quad \textit{States, Num} \mid \textit{States}, \ [\![\ldots]\!]$$

The **FR** operator represents the average firing rate of a certain transition, *i.e.* the average occurrence of a certain action.

$$\textit{FR} \quad ::= \quad \textit{Actions}$$

The **InInterval** operator determines whether a numerical value is within a certain interval or within multiple intervals. It returns a truth value.

$$\begin{aligned}\textit{InInterval} \quad ::= \quad & [\, \textit{ProbInInterval} \mid \textit{Moment} \mid \textit{FR} \mid \oplus \\ & \mid \textit{SS:P} \mid \textit{ProbInStates}\,] \times [[\![\ldots]\!]]^{1..*}\end{aligned}$$

The **InStates** operator is responsible for returning a truth value that expresses whether a certain state or set of states is included in or corresponds to another set of states.

$$\textit{InStates} \quad ::= \quad \textit{States, States}$$

The **ProbInInterval** operator returns the probability with which the passage takes place in a certain amount of time, defined by a time range.

$$\textit{ProbInInterval} \quad ::= \quad \textit{PTD}, \ [\![\ldots]\!]^{1..*} \mid \textit{Conv}, \ [\![\ldots]\!]^{1..*}$$

The **ProbInStates** operator corresponds to the transient probability of being in a certain set of states at a given instant in time, having started from a particular set of states. The first argument is the set of start states, the second the set of target states and the third the time instant of interest.

$$\textit{ProbInStates} \quad ::= \quad \textit{States, States, Num}$$

The **StatesAtTime** operator returns the set of states that the system can occupy at a given time instant with a certain probability. The first argument represents the time instant and the second the probability value or range.

$$\textit{StatesAtTime} \quad ::= \quad \textit{Num, Num} \mid \textit{Num}, \ [\![\ldots]\!]$$

4.1.2 Value Nodes

The $[\![\ldots]\!]$ node represents a range / interval. It has two arguments, both of which have a numerical value.

$$\begin{aligned}[\![\ldots]\!] \quad ::= \quad & ([\ \oplus \mid \textit{Num} \mid \textit{Moment} \mid \textit{ProbInInterval} \mid \\ & \textit{ProbInStates} \mid \textit{SS:P} \mid \textit{FR}] \times (\oplus \mid \textit{Num})) \\ & \mid \textit{Moment, Moment} \mid \textit{SS:P, SS:P} \mid \textit{FR, FR} \\ & \mid \textit{ProbInInterval}, \textit{ProbInInterval} \mid \textit{ProbInStates}, \\ & \textit{ProbInStates}\end{aligned}$$

The **States** node represents a set of states. The first annotation identifies the set of states either through state labels or by referencing states directly, and the second annotation is a label representing the type of the states. Labels identify states by specifying conditions on the model. We have two sets of atomic propositions, *SAL* = {state and action labels} and *TYP* = $\{start, \ target, \ incl., \ excl., \ time, \ prob., \ reward, \ moment, \ \emptyset\}$. We also have *State* $::= a \mid tt \mid$ *State* $\wedge$ *State* $\mid \neg$*State*, where $a \in$ *SAL*, and *Type* $\in$ *TYP*.

$$\textit{States} \quad \sim \quad \textit{State, Type}$$

The **Actions** node represents a set of actions. The first annotation identifies individual actions either through a set of labels or by referencing them directly. The second represents the type of action. We have *Action* $::= a \mid tt \mid$ *Action* $\wedge$ *Action* $\mid \neg$*Action*, where $a \in$ *SAL* and $Type \in$ *TYP*.

$$\textit{Actions} \quad \sim \quad \textit{Action, Type}$$

The **Num** node represents a numerical value. The first annotation is the numerical value itself, while the second identifies the type of the numerical value. We have that *Integer* $\in \mathbb{Z}$, *Real* $\in \mathbb{R}$ and *Type* $\in$ *TYP*.

$$\textit{Num} \quad \sim \quad \textit{Integer, Type} \mid \textit{Real, Type}$$

The **Bool** node represents a truth value. Its annotation is the truth value itself. We have that *Boolean* $\in \{true, \ false\}$.

$$\textit{Bool} \quad \sim \quad \textit{Boolean}$$

4.2 Type System

For the type system, the following basic types are used:

num	:	a numerical value
range	:	a range of numerical values
bool	:	a truth value
func	:	a distribution or density function
states	:	states of the system
actions	:	actions that can occur in the system

The type system for the operators used in the formalism is defined as follows:

$$
\begin{aligned}
? &\vdash (\mathit{bool} \mid \mathit{num} \mid \mathit{func} \mid \mathit{states})^{1..*} \\
; &\vdash (\mathit{bool} \mid \mathit{num} \mid \mathit{func} \mid \mathit{states})^{2..*} \\
\curlyvee &\vdash \mathit{bool}, \mathit{bool} : \mathit{bool} \\
\neg &\vdash \mathit{bool} : \mathit{bool} \\
\bowtie &\vdash \mathit{num}, \mathit{num} : \mathit{bool} \\
\oplus &\vdash \mathit{num}, \mathit{num} : \mathit{num} \\
\mathit{PTD} &\vdash \mathit{states}^{2..4} \mid \mathit{states}^{2..4}, \mathit{actions}^{1..2} \mid \\
&\quad \mathit{states}^{2..4}, \mathit{range} \mid \mathit{states}^{2..4}, \\
&\quad \mathit{actions}^{1..2}, \mathit{range} : \mathit{func} \\
\mathit{Dist} &\vdash \mathit{func} : \mathit{func} \\
\mathit{Conv} &\vdash \mathit{func}, \mathit{func} : \mathit{func} \\
\mathit{Moment} &\vdash \mathit{num}, \mathit{func} : \mathit{num} \\
\mathit{SS{:}P} &\vdash \mathit{states} : \mathit{num} \\
\mathit{SS{:}S} &\vdash \mathit{states}, \mathit{num} \mid \mathit{states}, \mathit{range} : \mathit{states} \\
\mathit{FR} &\vdash \mathit{actions} : \mathit{num} \\
\mathit{InInterval} &\vdash \mathit{num}, \mathit{range}^{1..*} : \mathit{bool} \\
\mathit{InStates} &\vdash \mathit{states}, \mathit{states} : \mathit{bool} \\
\mathit{ProbInInterval} &\vdash \mathit{func}, \mathit{range}^{1..*} : \mathit{num} \\
\mathit{ProbInStates} &\vdash \mathit{states}, \mathit{states}, \mathit{num} : \mathit{num} \\
\mathit{StatesAtTime} &\vdash \mathit{num}, \mathit{num} \mid \mathit{num}, \mathit{range} : \mathit{states} \\
[\![\ldots]\!] &\vdash \mathit{num}, \mathit{num}
\end{aligned}
$$

4.3 Examples

Performance Trees are best appreciated when demonstrated on specific examples. The performance queries introduced earlier are ideal candidates for the visualisation of the concept.

A good example of passage time requirements is Query 3 (cf. Section 3.1), which is seeking for a truth value that determines whether a passage between two sets of states occurs within a specified interval of time with probability lying in a given range.

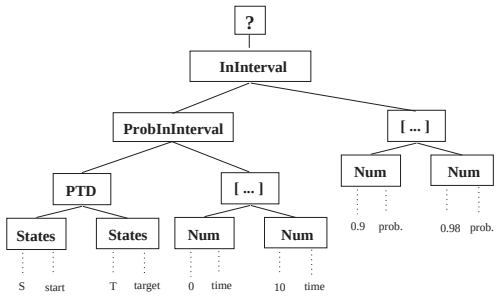


Figure 1. Query 3

In terms of transient requirements, Queries 5 and 6 are relevant. Query 5 aims at finding the probability of the system occupying a set of states at a given time instant, while Query 6 is interested in the set of states that the system occupies at a given time with a given probability.

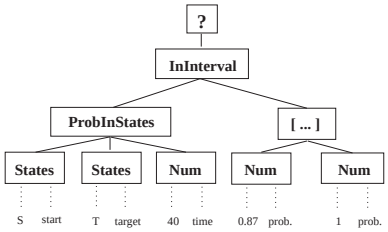


Figure 2. Query 5

Interesting requirements in terms of steady-state are contained in Queries 8 and 9. Query 8 is interested in obtaining a set of states having a certain steady-state probability, while Query 9 is looking at system productivity, obtained through transition firing rates.

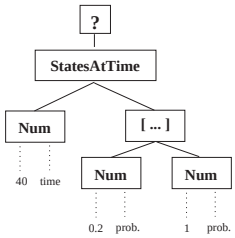


Figure 3. Query 6

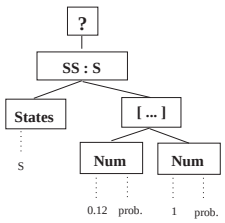


Figure 4. Query 8

Higher order requirements worth looking at are to be found in Queries 10 and 12. Query 10 addresses the variance of a passage time, taking into account included and excluded actions along the passage, while Query 12 is searching for an average value for the passage time, where the passage is composite, defined by a convolution of two independent passage time densities.

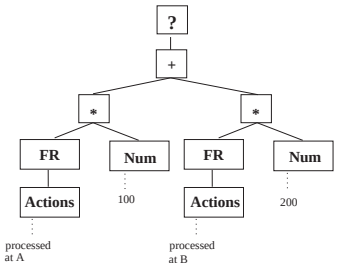


Figure 5. Query 9

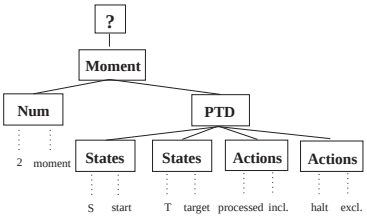


Figure 6. Query 10

5 Performance Trees vs. Stochastic Logics

As we have demonstrated, the Performance Tree formalism is well-equipped for specifying a large variety of system performance-related queries. The advantage of using Performance Trees as a representation mechanism lies in their broad expressiveness and the fact that they can be applied to general stochastic systems, without the need to rely on any underlying modelling methodology in particular. A further appealing feature is that they can not only be used for the verification of properties on stochastic models, but also for the extraction of valuable measures from them (such as obtaining passage time distributions and densities, higher moments, firing rates of actions, *etc.*).

In contrast, stochastic logics have been used successfully for some time now and have subsequently firmly established themselves in the performance community as the representation formalism of choice for stochastic model checking. Even though they enjoy such popularity, we believe that current stochastic logics lack a certain degree of expressiveness that would in many cases be desirable. It is possible to introduce extensions to existing logics or to merge them into new, possibly more powerful formalisms, in order to extend the range of requirements that can be catered for, but we regard Performance Trees as a more powerful and more convenient alternative. In addition, most of the underlying mathematical techniques and tools necessary for evaluating queries expressed as Performance Trees already exist. Also, due to the ability to depict performance query representations graphically, in a way that reflects the logical structure of natural language performance queries, Performance Trees can be understood and used perhaps more

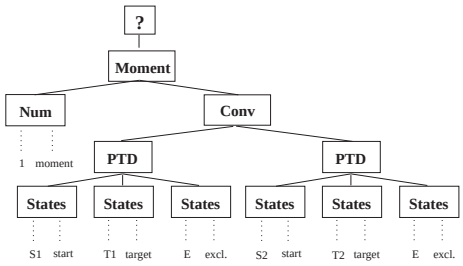


Figure 7. Query 12

naturally than stochastic logic formulae.

The Performance Tree formalism subsumes CSL, since it is capable of expressing everything that CSL can. However, to emphasise the differences in terms of expressiveness between the two formalisms, we give a brief list of the types of requirements that stochastic logics are unable to represent, but which Performance Trees were designed to express: distributions, densities and convolutions, higher moments, firing rates of transitions, arithmetic operations, included states and excluded actions along a passage and multiple probability and time intervals. To show that Performance Trees do indeed subsume CSL, we present below how the mapping of CSL model checking questions onto Performance Trees is carried out.

A CSL model checking question has the form $s \models \sigma$. The purpose of model checking is to verify whether the satisfaction relation holds for a particular state s and formula σ. For the mapping, we employ the function $f(s,x)$, where s is a state and x is a CSL formula. f translates a CSL model checking question into the corresponding Performance Tree representation.

$$
\begin{aligned}
f(s,tt) &= ?(tt)\\
f(s,a) &= ?(\mathit{InStates}(\mathit{States}(a), \mathit{States}(\mathcal{L}(s)))\\
f(s,\neg\sigma) &= ?(\neg(f(s,\sigma)))\\
f(s,\sigma_1 \wedge \sigma_2) &= ?(\wedge(f(s,\sigma_1),\ f(s,\sigma_2)))\\
f(s,\mathcal{S}_{\bowtie p}(\sigma)) &= ?(\bowtie (\\
&\quad \mathit{SS{:}P}(\mathit{States}(\{x : f(x,\sigma)\})),\\
&\quad \mathit{Num}(p, \mathit{prob.})))\\
f(s,\mathcal{P}_{\bowtie p}(\sigma_1 \mathcal{U}^I \sigma_2)) &= ?(\bowtie (\\
&\quad \mathit{ProbInInterval}(\\
&\quad\ \mathit{PTD}(\\
&\quad\quad \mathit{States}(\{s\}, \mathit{start}),\\
&\quad\quad \mathit{States}(\{x : f(x,\neg\sigma_1)\}, \mathit{excl.}),\\
&\quad\quad \mathit{States}(\{x : f(x,\sigma_2)\}, \mathit{target})),\\
&\quad\ g(I)),\\
&\quad \mathit{Num}(p, \mathit{prob.})))
\end{aligned}
$$

where $a \in \mathit{SAL}$ and $\mathcal{L} : S \rightarrow 2^{\mathit{SAL}}$ is a labelling func-

tion that assigns to a state in the finite set of states S a label from *SAL*, the set of atomic propositions. $g(I)$ is a function that converts the interval I into the range representation of Performance Trees, such that if $I = [x, y]$ then $g(I) = [...](\mathit{Num}(x),\ \mathit{Num}(y))$. The Next operator $\mathcal{X}^I\sigma$ can be expressed with the Until operator as $\mathcal{X}^I\sigma \equiv tt\,\mathcal{U}^I\sigma$, therefore there is no need to have a specific translation rule.

6 Worked Example: Voting system

Below, we present an example of a voting system from [7] with possible breakdowns and a recovery mechanism to illustrate the applicability of the Performance Tree formalism to real-life systems. The voting system is modelled by an SM-SPN, as shown in Figure 8.

In order to help specify the underlying semi-Markov process, the transition specification table on the right of Figure 8 displays the (selection weight, priority, firing distribution) triples associated with the transitions in the SM-SPN. From the model, we can see that a voter can only be processed if a polling unit is available and that the vote can only be counted if a voting server is available. Once a vote has been processed, the polling unit that dealt with the voter casting the vote is freed up, as is the voting server that processed the vote. We now present a performance query that is to be executed on the model:

V1. *"Can we be* 98% *confident that the voting system will be in state* 'all voters have voted' *at time instant* 730*, given that it started in state* 'no voters have voted yet' *at time* 0*?"*

This is a passage-related query, so we need to specify start and target states. We do that by creating relevant labels at the Petri net level, since states (and actions) are easily identified in this way. We use S as the label for the set of start states and T as the label for the set of target states. For the sake of simplicity, we substitute S for *'no voters have voted yet'* and T for *'all voters have voted'*. For labelling purposes, a table is maintained for every model, each entry of which is a tuple consisting of an identifier and a constraint. The identifier is a label that stands for a particular set of reachable states in the model. The model-level identification of this set of states is achieved by the constraint, which needs to be satisfied. In other words, the set of states that satisfies the constraint is associated with the label. In our example, S and T are defined as follows:

$$(S, (not\_voted, CC)) \qquad (T, (voted, CC))$$

A constraint is dependent on the underlying model, but in this particular case, it concerns the number of tokens on the places in the net, since their flow indicates change in the system. Having defined the necessary labels, we can formulate our query as a Performance Tree as shown in Figure 9.

One method that can be used for resolving the query is the

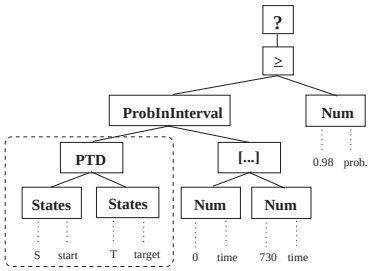


Figure 9. Query V1: extraction of a passage time quantile constraint from a density function

response time analysis method based on numerical Laplace transform inversion, as described in [8]. When analysing the voting system with 300 voters in the model, we obtain a state space of 10.9 million states. Figure 10 shows the overlap of the density function calculated analytically and by simulation for the time taken to process all voters. The Performance Tree specification for this passage time density is given by the boxed subtree shown in Figure 9. This passage specification is incorporated into the complete Query V1 to give the results depicted in Figure 11, which shows the cumulative distribution function of the same passage, as well as the quantile of the time taken to process all voters in the system by time 730. We present a further Performance Tree query as an instance of a measure that could not be expressed in CSL:

V2. *"What is the expected time until two servers have broken down in the voting system?"*

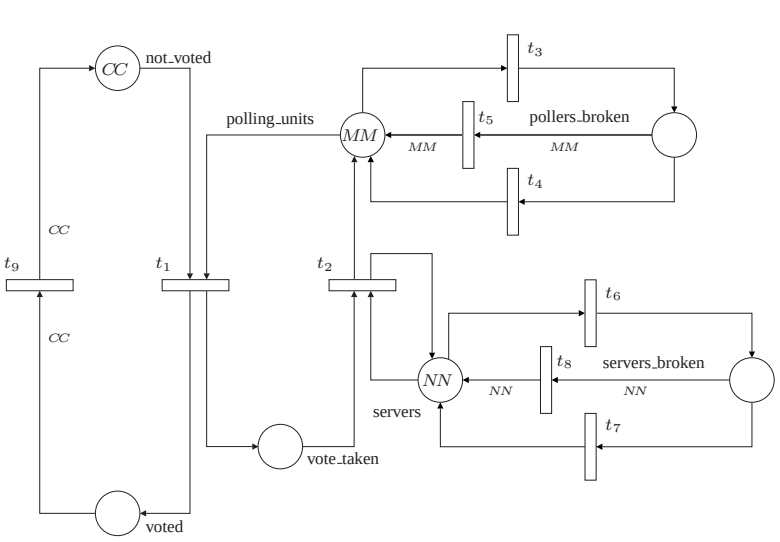
As before, this is a passage-related query, so we need to create labels S and T to define the start and target states of the passage.

$$\begin{array}{ll} (S, & (not\_voted, CC) \ \wedge\ (servers, NN) \wedge \\ & (polling\_units, MM)) \\ (T, & (server\_broken, 2)) \end{array}$$

Having defined the necessary labels, we can formulate our query as a Performance Tree as shown in Figure 12. Higher moments could be extracted as required by the modeller.

7 Conclusions and Future Work

In this paper, we have presented the Performance Tree framework, which seeks to expand the present boundaries of performance query specification. Performance Trees introduce a new environment for the verification of performance requirements and the quantitative analysis of performance measures on stochastic models.



SM-SPN Transition specification

$t_1 \rightarrow (1.0, 1, \exp(0.5, s))$
$t_2 \rightarrow (1.0, 1, \det(0.01, s))$
$t_3 \rightarrow (0.05, 1, \exp(1.5, s))$
$t_4 \rightarrow (1.0, 1, \mathrm{uni}(1.0, 2.0, s))$
$t_5 \rightarrow (1.0, 2, 0.8\,\mathrm{uni}(1.5, 10, s) + 0.2\,\mathrm{erl}(0.2, 5, s))$
$t_6 \rightarrow (0.01, 1, \exp(1.0, s))$
$t_7 \rightarrow (1.0, 1, \mathrm{uni}(2.0, 3.0, s))$
$t_8 \rightarrow (1.0, 2, \mathrm{gam}(0.1, 12.337, s))$
$t_9 \rightarrow (1.0, 2, \det(0, s))$

Figure 8. A voting system model with breakdowns and recoveries

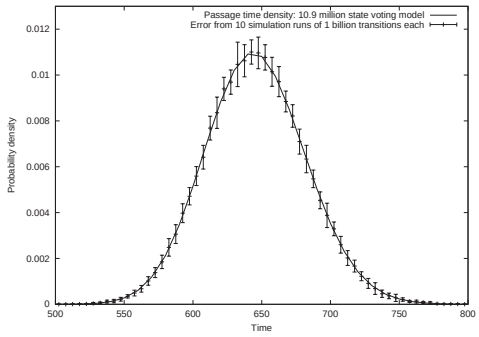


Figure 10. Density function for completed voting passage time in the voting model.

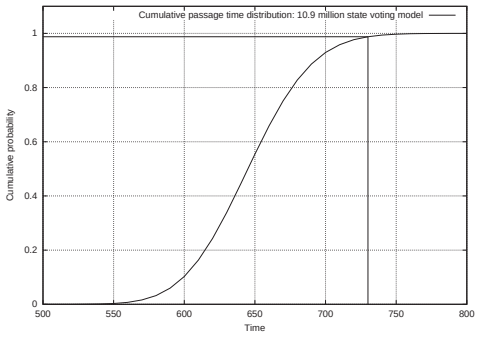


Figure 11. Cumulative distribution function for voting model. Probability of having completed voting by time 730 is 0.9876.

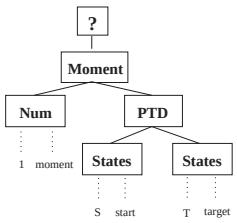


Figure 12. Query V2: a moment calculation for a passage time density

The main reason for this is the extended expressiveness of the formalism, which allows us to formulate a multitude of performance queries that were previously not formally expressible. The relative ease of constructing these queries is an important factor that sets Performance Trees apart from other formalisms, which are mostly based on obfuscating logical expressions that require an expert understanding for effective application.

Since most existing model checkers are based on stochastic logics, it is clear that no one existing tool can currently resolve the full range of queries expressible in our new representation. However, we anticipate no difficulties for existing tools to incorporate an interface for Performance Trees. This would facilitate the implementation of a "meta-tool" environment with a complete analysis capability. Alternatively, since algorithms used in these tools are accessible via public literature, a more ambitious goal

would be to construct a large open-source integrated analysis toolset. Such a system will also give rise to new opportunities for the integration of previously unexploited methods for the parallelisation, distribution and optimisation of computation in model checking and quantitative analysis.

Below is a diagram of the proposed workflow from the initial composition of a performance query to the final result obtained from its verification on the model. Work targeted

at developing tool support for the design and subsequent analysis of Performance Trees is in progress, forming part of the GRAIL[1] project. Future publications will detail the applicability of Performance Trees to a variety of modelling formalisms, present formal semantics and discuss appropriate underlying mathematical analysis methods.

References

[1] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[2] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous-time Markov chains. In *Computer-Aided Verification*, volume 1102 of *LNCS*, pages 269–276, 1996.

[3] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[4] C. Baier, L. Cloth, B. R. Haverkort, M. Kuntz, and M. Siegle. Model checking action- and state-labelled Markov chains. *DSN'04, Proc. Intl. Conf. on Dependable Systems and Networks*, pages 701–710, June 2004.

[5] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisations of performability properties. In *Proc. ICALP 2000*, volume 1853 of *LNCS*, pages 780–792, 2000.

[6] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, June 2003.

[7] 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, Proc. Petri Nets and Performance Models*, pages 62–71, University of Illinois at Urbana-Champaign, September 2003.

[1] "Grid-enabled Performance Analysis using Stochastic Logics". See `http://aesop.doc.ic.ac.uk/projects/grail` for more details. The project is supported by EPSRC grant EP/D505933/1.

[8] 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. *Journal of Linear Algebra and Applications*, 386:311–334, July 2004.

[9] P. Buchholz, J.-P. Katoen, P. Kemper, and C. Tepper. Model-checking large structured Markov chains. *Journal of Logic and Algebraic Programming*, 56:69–96, 2003.

[10] G. Ciardo, J. K. Muppala, and K. S. Trivedi. SPNP: Stochastic Petri Net Package. In *PNPM'89, Proc. 3rd Intl. Workshop on Petri Nets and Performance Models*, pages 142–151, 1989.

[11] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius modeling tool. In B. Haverkort and R. German, editors, *Proceedings the 9th International Workshop on Petri Nets and Performance Models*, pages 241–250. IEEE Computer Society Press, Aachen, September 2001.

[12] H. Hermanns. *Interactive Markov Chains*. PhD thesis, Universität Erlangen-Nürnberg, July 1998.

[13] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. Towards model checking stochastic process algebra. In *IFM 2000, Proc. 2nd Intl. Conf. on Integrated Formal Methods*, pages 420–439, November 2000.

[14] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A tool for model checking Markov chains. *Software Tools for Technology Transfer*, 4(2):153–172, 2002.

[15] J. Hillston. *A Compositional Approach to Performance Modelling*, volume 12 of *Distinguished Dissertations in Computer Science*. 1996.

[16] C. Hirel, R. Sahner, X. Zhang, and K. S. Trivedi. Reliability and performability modeling using SHARPE 2000. In *TOOLS 2000, Proc. 11th Intl. Conf. on Computer Performance Evaluation, Modelling Techniques and Tools*, volume 1786 of *LNCS*, page 345, 2000.

[17] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *QEST'05, Proc. 2nd Intl. Conf. on the Quantitative Evaluation of Systems*, pages 243–244, Italy, September 2005.

[18] W. J. Knottenbelt. Generalised Markovian analysis of timed transitions systems. M.Sc. thesis, University of Cape Town, South Africa, July 1996.

[19] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *TACAS'02, Proc. Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 52–66, 2002.

[20] J. Matthes. Zur Theorie der Bedienungsprozesse. In *Transactions of the 3rd Prague Conference on Information Theory, Statistical Decision Functions and Random Processes*, pages 513–528, 1962.

[21] R. Pyke. Markov renewal processes: Definitions and preliminary properties. *Annals of Mathematical Statistics*, 32(4):1231–1242, December 1961.