Probabilistic Verification at Runtime for Self-Adaptive Systems

Antonio Filieri and Giordano Tamburrelli

DeepSE Group @ DEI - Politecnico di Milano, Italy
{ filieri | tamburrelli }@elet.polimi.it

Abstract. An effective design of effective and efficient self-adaptive systems may rely on several existing approaches. Software models and model checking techniques at run time represent one of them since they support automatic reasoning about such changes, detect harmful configurations, and potentially enable appropriate (self-)reactions. However, traditional model checking techniques and tools may not be applied as they are at run time, since they hardly meet the constraints imposed by on-the-fly analysis, in terms of execution time and memory occupation. For this reason, efficient run-time model checking represents a crucial research challenge.
This paper precisely addresses this issue and focuses on probabilistic run-time model checking in which reliability models are given in terms of Discrete Time Markov Chains which are verified at run-time against a set of requirements expressed as logical formulae. In particular, the paper discusses the use of probabilistic model checking at run-time for self-adaptive systems by surveying and comparing the existing approaches divided in two categories: state-elimination algorithms and algebra-based algorithms. The discussion is supported by a realistic example and by empirical experiments.

1 Introduction

Software is the driving engine of modern society. Most human activities, including the most critical ones, are either software enabled or entirely managed by software. As software is becoming ubiquitous and society increasingly relies on it, the adverse impact of unreliable or unpredictable software cannot be tolerated. Indeed, software systems have to be able to evolve correspondingly to their deployment environment in order to guarantee a seamless fulfillment of desired requirements and ensure a minimal downtime. In response to this challenge, current Software Engineering aims at designing *Self-Adaptive Systems* which are able to react and reconfigure themselves minimizing human intervention and ideally guaranteeing a lifelong requirement fulfillment. To date, Software Engineering research in self-adaptive systems has produced promising initial results, as illustrated for example in [24]. However, even if these findings provide an essential step towards a set of effective and efficient solutions for self-adaptation, they are not the end of the story as building these dependable systems is still unclear and requires further investigation.

Designers must ensure that any critical requirement of the system continues to be satisfied before, during and after unforeseen scenarios. By this we mean that software systems are required to be *dependable*, to avoid damaging effects due to violated requirements that can range from loss of business to loss of human lives. At the same time, the complexity of modern software systems has grown enormously in the past years with users always demanding for new features and better quality of service. Software systems changed from being monolithic and centralized to modular, distributed, and dynamic. They are increasingly composed of heterogeneous components and infrastructures on which software is configured and deployed. When an application is initially designed, software engineers often only have a partial and incomplete knowledge of the external environment in which the application will be embedded at run time. Design may therefore be subject to high uncertainty. This is further exacerbated by the fact that the structure of the application, in terms of components and interconnections, often changes dynamically. New components may become available and published by providers for use by potential clients. Some components may disappear, or become obsolete, and new ones may be discovered dynamically. This may happen, for example, in the case of Web service-based systems [6,7]. This also happens in pervasive computing scenarios where devices that run application components are mobile [27]. Because of mobility, and more generally context change, certain components may become unreachable, while others become visible during the application's lifetime. Finally, requirements also change continuously and unpredictably, in a way that is hard to anticipate when systems are initially built. Because of uncertainty and continuous external changes the software application is subject to continuous adaptation and evolution.

This paper focuses on how analyzing and comparing existing approaches aimed at managing run-time changes by verifying that the software evolves dynamically without disrupting the dependability of applications. Dependability includes attributes such as reliability, availability, performance, safety, security. In this paper we focus our attention on two main dependability requirements that typically arise in the case of decentralized and distributed applications: namely, *reliability* and *performance*. Both reliability and performance depend on environment conditions that are hard to predict at design time, and are subject to a high degree of uncertainty. For example, performance may depend on end-user profiles, on network congestion, on load conditions of external services that are integrated in the application. Similarly, reliability may depend on the behavior of the network and of the external services that compose the application being built.

Existing approaches focus on supporting the development and operation of complex and dynamically evolvable software systems leveraging on *Formal Models*. These formal models are built at design time to support an initial assessment that the application satisfies the requirements. Models are then kept alive at run time and continuously verified to check that the changes with respect to the design-time assumptions do not lead to requirements violations. This requires efficient mechanisms for run-time verification. If requirements violations are detected, appropriate actions must be undertaken, ranging from off-line evolution to on-line adaptation. In particular, much research is currently investigating the extent to which the software can respond to predicted or detected requirements violation through self-managed reactions, in an autonomic manner. These, however, are out of the scope of this paper, which only focuses on describing run-time verification approaches.

Verification at runtime of reliability and performance properties for self-adaptive systems typically relies on *Probabilistic Models* such as: *Discrete Time Markov Chains* and *Discrete Time Markov Rewards Models*. This paper introduces these formalisms and subsequently illustrates and compares the approaches for efficient runtime verification. In particular, our contribution is structured as follows. Section 2 describe the mathematical foundations of Markov Chains and PCTL (i.e., the logic commonly adopted to verify properties on discrete Markov models). Such mathematical concepts are described by relying on a realistic example also introduced in this section and used throughout the paper to illustrate the different model checking techniques. Section 3 dicusses and compares the existing approaches for run-time verification. Finally, 5 draws some remarking conclusions and illustrates potential future work.

2 Probabilistic Models for Run-time Verification

This section provides an introduction to the probabilistic models adopted to express reliability and performance properties for self-adaptive systems. In this section we provide a brief introduction to the mathematical concepts used throughout the paper. In particular in Sections 2.3 and 2.4 we describe respectively the *Probabilistic Computational Tree Logic* and its extension with rewards used to represent properties of systems to be verified at runtime. The reader can refer to [4,5] for a comprehensive in-depth treatment of these concepts.

2.1 Discrete Time Markov Chains

Discrete Time Markov Chains (DTMCs) are a widely accepted formalism to model reliability of systems built by integrating different components. In particular, they proved to be useful for an early assessment or prediction of reliability [21]. The adoption of DTMCs implies that the modeled system meets, with some tolerable approximation, the Markov property–described below. This issue can be easily verified as discussed in [8,14].

DTMCs are discrete stochastic processes with the Markov property, according to which the probability distribution of future states depends only upon the current state. They are defined as a Kripke structure with probabilistic transitions among states. *States* represent possible configurations of the system. *Transitions* among states occur at discrete time and have an associated probability.

Formally, a (labeled) DTMC is a tuple $(S, S_0, \mathbf{P}, L)$ where

– S is a finite set of states

- $S_0 \subseteq S$ is a set of initial states
- $\mathbf{P} : S \times S \rightarrow [0,1]$ is a stochastic matrix ($\forall s \in S \mid \sum_{s' \in S} \mathbf{P}(s,s') = 1$). An element $\mathbf{P}(s_i, s_j)$ represents the probability that the next state of the process will be s_j given that the current state is s_i.
- $L : S \rightarrow 2^{AP}$ is a labeling function. AP is a set of atomic propositions. The labeling function associates to each state the set of atomic propositions that are true in that state.

A state $s \in S$ is said to be an *absorbing state* if $\mathbf{P}(s,s) = 1$ otherwise the state is a *transient state*. If a DTMC contains at least one absorbing state, the DTMC itself is said to be an *absorbing DTMC*. We further assume that in our models for any transient state there is a non zero probability of reaching at least one of the absorbing states. In the simplest model for reliability analysis, the DTMC will have two absorbing states, representing the correct accomplishment of the task and the task's failure, respectively. The use of absorbing states is commonly extended to modeling different failure conditions. For example, different failure states may be associated with the invocation of different external services. The set of failures to look for is strictly domain-dependent.

In an absorbing DTMC with r absorbing states and t transient states, rows and columns of the transition matrix $\mathbf{P}$ can be reordered such that $\mathbf{P}$ is in the following *canonical form*:

$$\mathbf{P} = \begin{pmatrix} Q & R \\ 0 & I \end{pmatrix} \tag{1}$$

where I is an r by r identity matrix, 0 is an r by t zero matrix, R is a nonzero t by r matrix and Q is a t by t matrix. Consider now two distinct transient

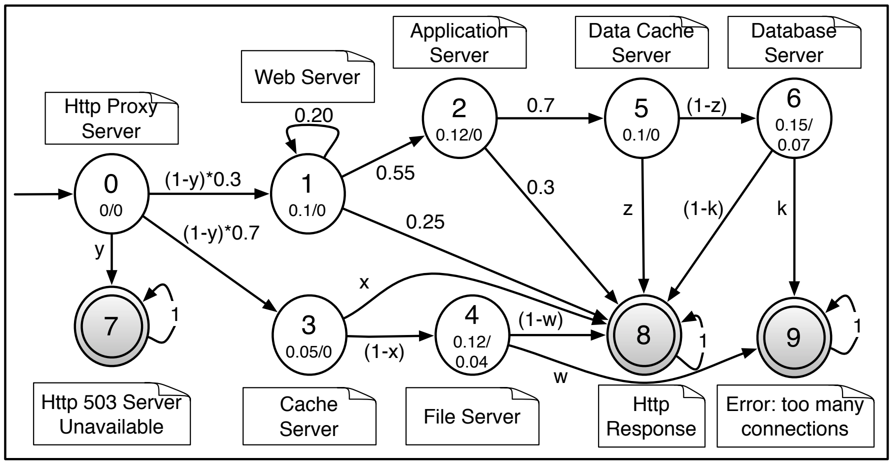


Fig. 1. DTMC Example

states s_i and s_j. The probability of moving from s_i to s_j in exactly 2 steps is $\sum_{s_x \in S} P(s_i, s_x) \cdot P(s_x, s_j)$. Generalizing, for a k-steps path and recalling the definition of matrix product, it follows that the probability of moving from any transient state s_i to any other transient state s_j in exactly k steps corresponds to the entry (s_i, s_j) of the matrix Q^k. As a natural generalization, we can define Q^0, which represents the probability of moving from each state s_i to s_j in 0 steps, as the identity t by t matrix, whose elements are 1 iff $s_i = s_j$ [15].

Due to the fact that R must be a nonzero matrix, and $\mathbf{P}$ is a stochastic matrix, Q has uniform-norm strictly less than 1, thus $Q^n \to 0$ as $n \to \infty$, which implies that eventually the process will be absorbed with probability 1.

Let us consider for example the DTMC in Figure 1, which represents a typical web architecture. The system comprises an HTTP Proxy server, a Web server and an application server. In addition, structured data and static content (e.g., files, images, etc.) are respectively stored in a Database server and File server. Both of them are cached by ad-hoc cache servers. Each state is labelled by a numeric label and by a couple in the form n_1/n_2, its meaning will be clear later on. States 7, 8 and 9 are absorbing states. The former represents the failure of serving an incoming request due to an unavailable server (e.g., overloaded server or maintenance downtime). The latter represent the endpoint of a correct HTTP request. Transitions among non-absorbing states reports the probability for an HTTP request of passing from one element of the architecture to the other. For example transition $0 - 1$ indicates the probability that a request is associated to static or dynamic content. Transition $1 - 1$ indicates instead the probability of an HTTP self-redirect.

Conversely, transitions to absorbing states indicates the final outcome in processing a request. We use variables as transition labels to indicate that the value of the corresponding probability is unknown, and may change over time. For example transitions $3 - 4$ and $5 - 6$ indicate the cache hit probability.

In matrix form, the same DTMC would be characterized by the following matrices Q and R:

$$Q = \begin{pmatrix} 0 & (1-y)0.3 & 0 & (1-y)0.7 & 0 & 0 & 0 \\ 0 & 0.2 & 0.55 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.7 & 0 \\ 0 & 0 & 0 & 0 & 1-x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-z \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} y & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0.3 & 0 \\ 0 & x & 0 \\ 0 & 1-w & w \\ 0 & z & 0 \\ 0 & 1-k & k \end{pmatrix}$$

Notice that the parameters necessary to model a system with a DTMC (i.e., the values in the DTMC matrix) may be obtained by experimental results as well as by estimates extracted by similar systems or by previous version of the system under design. Finally, the system reported in this section is just a toy example that we use to introduce the proposed approach. However, the concepts described hereafter apply seamlessly to real systems which might have thousands of states and failures.

2.2 Discrete Time Markov Rewards Models

A D-MRM [1] is a DTMC augmented with rewards, through which one can quantify a benefit (or loss) due to the residence in a specific state or the move along a certain transition. A D-MRM has an underlying DTMC, through which designers can provide a high-level model for the system's control flow by abstracting the execution state space into a finite set of abstract states relevant to the verification[1]. As illustrated later on, D-MRM may be used to model performance properties or even properties concerning costs (e.g., energy consumptions).

A reward is a non-negative value assigned to a state or a transition. Rewards can represent information such as average execution time, power consumption, number of I/O operations, or even cost of an outsourced operation. A D-MRM is a tuple (S, S_0, P, L, ρ, i) where:

- S is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $P : S \times S \to [0, 1]$ is a stochastic matrix ($\forall s \in S \mid \sum_{s' \in S} P(s, s') = 1$). An element $P(s_i, s_j)$ represents the probability that the next state of the process will be s_j given that the current state is s_i,
- $L : S \to 2^{AP}$ is a labeling function which assigns to each state the set of *Atomic Propositions* that are true in the state,
- $\rho : S \to \mathbb{R}_{\geq 0}$ is a *state reward* function assigning to each state a non-negative real number,
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ is a *transition reward* function assigning a non-negative real number to each transition.

To understand how rewards are gained, we need to precisely state how the system modeled by the D-MRM evolves over a sequence of time steps. At step 0 the system enters the initial state s_0. At step 1, the system gains the reward $\rho(s_0)$ associated with the initial state and moves to a new state (say, $s1$), gaining also the reward $\iota(s_0, s_1)$. The cumulated reward when the system enters state $s1$ is $\rho(s_0) + \iota(s_0, s_1)$. At step 2, it gains the reward $\rho(s_1)$ associated with state $s1$, and then exits it gaining also the reward associated with the chosen transition, and so on. In summary, the state reward is acquired if the D-MRM resides in state s_i for one time step. The reward associated with a transition $\iota(s_i, s_j)$ is

[1] The adoption of an underlying Markov model implies that the modeled system meets, with some tolerable approximation, the Markov property, according to which the probability distribution of future states depend only on the current state.

gained as the process makes an instantaneous move from state s_i to state s_j. A state $s \in S$ is said to be an *absorbing state* if $P(s,s) = 1$. If a D-MRM contains at least one absorbing state, the D-MRM itself is said to be an *absorbing D-MRM*. If the absorbing states are reachable, in any number of time steps, from transient ones, it can be shown that any execution will eventually be absorbed with probability 1 (as proved for DTMCs in [29]). We assume D-MRMs to be well-formed, i.e. all states are reachable from the initial state and for all non absorbing states it is possible to reach a least one absorbing state.

Transitions can be defined through a matrix P where $P(s_i, s_j)$ represents the probability associated with the transition from state s_i to state s_j. Let us now consider two distinct states s_i and s_j. The probability of moving from s_i to s_j in 2 steps is $\sum_{s_x \in S} P(s_i, s_x) \cdot P(s_x, s_j)$. Generalizing to a k-steps path and recalling the definition of matrix product, the probability of moving from any state s_i to any other state s_j in k steps corresponds to the entry (s_i, s_j) of the matrix P^k. As a natural generalization, we can define P^0 (representing the probability of moving from a state s_i to a state s_j in 0 steps) as the identity matrix, whose elements are 1 iff $s_i = s_j$ [15,29].

Variability can be modeled quite simply in D-MRMs. We assume that variability does not affect the structure of the models, only parameters. In our case, it only affects the possible values used to label transition probabilities and rewards. This is usually expressive enough to accommodate changes in the environment that affect our system.

2.3 Probabilistic Computation Tree Logic

Formal languages to express properties of systems modeled through DTMCs have been studied in the past and several proposals are supported by model checkers to prove that a model satisfies a given property. In this paper, we focus on Probabilistic Computation Tree Logic (PCTL) [19,2], a logic that can be used to express a number of interesting reliability properties.

PCTL is defined by the following syntax:

$$\begin{aligned} \Phi &::= true \mid a \mid \Phi \wedge \Phi \mid \neg \Phi \mid \mathcal{P}_{\bowtie p}(\Psi) \\ \Psi &::= X\Phi \mid \Phi U^{\leq t} \Phi \end{aligned}$$

where $p \in [0,1]$, $\bowtie \in \{<, \leq, >, \geq\}$, $t \in \mathcal{N} \cup \{\infty\}$, and a represents an atomic proposition. The temporal operator X is called *Next* and U is called *Until*. Formulae generated from Φ are referred to as *state formulae* and they can be evaluated to either true or false in every single state, while formulae generated from Ψ are named *path formulae* and their truth is to be evaluated for each execution path.

The satisfaction relation for PCTL is defined for a state s as:

$$\begin{aligned}
&s \models true \\
&s \models a \quad \text{iff} \quad a \in L(s) \\
&s \models \neg\Phi \quad \text{iff} \quad s \not\models \Phi \\
&s \models \Phi_1 \wedge \Phi_2 \quad \text{iff} \quad s \models \Phi_1 \textit{ and } s \models \Phi_2 \\
&s \models \mathcal{P}_{\bowtie p}(\Psi) \quad \text{iff} \quad Pr(s \models \Psi) \bowtie p
\end{aligned}$$

A complete formal definition of $Pr(s \models \Psi)$ can be found in [5]; details are omitted here for simplicity. Intuitively, its value is the probability of the set of paths starting in s and satisfying Ψ.Given a path π, we denote its i-th state as $\pi[i]$; $\pi[0]$ is the initial state of the path. The satisfaction relation for a path formula with respect to a path π originating in s ($\pi[0] = s$) is defined as:

$$\begin{aligned}
&\pi \models X\Phi \quad \text{iff} \quad \pi[1] \models \Phi \\
&\pi \models \Phi_1 U^{\leq t} \Phi_2 \quad \text{iff} \quad \exists\, 0 \leq j \leq t \\
&(\pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j\, \pi[k] \models \Phi_1))
\end{aligned}$$

From the *Next* and *Until* operators it is possible to derive others. For example, the *Eventually* operator (often represented by the $\Diamond^{\leq t}$ symbol) is defined as:

$$\Diamond^{\leq t}\phi \;\equiv\; true\; U^{\leq t}\phi$$

It is customary to abbreviate $U^{\leq\infty}$ and $\Diamond^{\leq\infty}$ as U and $\Diamond$, respectively

PCTL can naturally represent reliability-related properties for a DTMC model of the application. For example, we may easily express constraints that must be satisfied concerning the probability of reaching absorbing failure or success states from a given initial state. These properties belong to the general class of *reachability properties*. Reachability properties are expressed as $\mathcal{P}_{\bowtie p}(\Diamond\ \Phi)$, which expresses the fact that the probability of reaching any state satisfying Φ has to be in the interval defined by constraint $\bowtie p$. In most cases, Φ just corresponds to the atomic proposition that is true only in an absorbing state of the DTMC. In the case of a failure state, the probability bound is expressed as $\leq x$, where x represents the upper bound for the failure probability; for a success state it would be instead expressed as $\geq x$, where x is the lower bound for success. PCTL is an expressive language through which more complex properties than plain reachability may be expressed. Such properties would be typically domain-dependent, and their definition is delegated to system designers.

Recalling our example of Figure 1, we may have the following reliability requirements:

- **R1**: *"The probability of successfully handling a request is greater than* 0.999*"*
- **R2**: *"The probability for a request of being dropped by the file server of the database server because of too many concurrent connections is less than* 0.001*"*
- **R3**: *"The probability for a request of experiencing an error HTTP 503 (e.g., too many incoming requests) is less than* 0.001*"*

- **R4**: *"The probability for a request of dynamic content of experiencing a cache miss is less than* 0.25*"*
- **R5**: *"The probability for a request of static content of experiencing a cache miss is less than* 0.15*"*

These requirements can be translated into PCTL as shown in Table 1, where the notation $s = n$ refers to the identification of state n according to Fig. 1. Notice that these requirements have different sets of initial states: **R1-3** must be evaluated starting from state 0 (i.e., $S_0 = \{0\}$) while **R4-5** must be evaluated starting respectively from state 1 and 3.

Table 1. Requirements translation in PCTL.

Req.	PCTL
R1	$P_{\geq 0.999}(true\ U\ s = 8) = P_{\geq 0.999}(\diamond\ s = 8)$
R2	$P_{\leq 0.001}(true\ U\ s = 9) = P_{\leq 0.001}(\diamond\ s = 9)$
R3	$P_{\leq 0.001}(X\ s = 7)$
R4	$P_{\leq 0.25}(true\ U\ s = 6)$ in $s = 1$
R5	$P_{\geq 0.15}(X\ s = 4)$ in $s = 3$

Given the formalisms explained so far, we can introduce in the next section the proposed approach which allow to efficiently verify non-functional requirements such as $R1 - 5$ at run-time via synthesis of symbolic expressions.

2.4 Extending PCTL with Rewards

R-PCTL is a logic language to express properties of a D-MRM. it is defined as follows [25]:

$$\Phi ::= true \mid a \mid \Phi\ \wedge\ \Phi \mid \neg\ \Phi \mid \mathcal{P}_{\bowtie p}\ (\Psi) \mid \mathcal{R}_{\bowtie r}\ (\Theta)$$
$$\Psi ::= X\ \Phi \mid \Phi\ U\ \Phi \mid \Phi\ U^{\leq t}\ \Phi$$
$$\Theta ::= I^{=k} \mid C^{\leq k} \mid\ \ \diamond \Phi$$

Formulae Φ are named *state formulae* and can be evaluated over a boolean domain (true, false) in each state. Formulae Ψ are named *path formulae* and describe a pattern that can be matched over the set of all possible paths originating in a given state. Symbol $\bowtie$ stands for a relational operator in the set $\{\leq, <, \geq, >\}$, $p \in [0, 1]$ is a probability bound, $r \in \mathbb{R}_{\geq 0}$, and $k \in \mathbb{Z}_{\geq 0}$. $trueU\Phi$ can be shortened by the *eventually* operator $\diamond\Phi$, with exactly the same semantics. The expressions defined by Θ support the specification of *reward patterns*.

Let us now informally discuss the semantics of R-PCTL, first ignoring reward formulae. The intuitive meaning of the formula $\mathcal{P}_{\bowtie p}(\Psi)$ evaluated in a state s, where Ψ is a path formula, is: the probability for the set of paths originating from s and satisfying $\varPsi$ meets the bound expressed as $\bowtie p$. More precisely, the satisfaction relation for (non-reward) state formulae is defined for a state s as:

$$\begin{array}{l}
s \models true \\
s \models a \quad \text{iff} \quad a \in L(s) \\
s \models \neg\varPhi \quad \text{iff} \quad s \not\models \varPhi \\
s \models \varPhi_1 \wedge \varPhi_2 \quad \text{iff} \quad s \models \varPhi_1 \; and \; s \models \varPhi_2 \\
s \models \mathcal{P}_{\bowtie p}(\varPsi) \quad \text{iff} \quad Pr(s \models \varPsi) \bowtie p)
\end{array}$$

A formal definition of how to compute $Pr(s \models \varPsi)$ is presented in [5]. The intuition is that its value corresponds to the probability of taking a path that satisfies $\varPsi$, among all the, possibly infinite, paths originating in s. The satisfaction relation for a path formula with respect to a path π originating in s ($\pi[0] = s$) is defined as:

$$\begin{array}{l}
\pi \models X\varPhi \quad \text{iff} \quad \pi[1] \models \varPhi \\
\pi \models \varPhi U \varPsi \quad \text{iff} \quad \exists j \geq 0.(\pi[j] \models \varPsi \wedge (\forall 0 \leq k < j.\pi[k] \models \varPhi)) \\
\pi \models \varPhi U^{\leq t} \varPsi \quad \text{iff} \quad \exists 0 \leq j \leq t.(\pi[j] \models \varPsi \wedge (\forall 0 \leq k < j.\pi[k] \models \varPhi))
\end{array}$$

Let us now focus on the semantics of the rewards fragment of R-PCTL. We intuitively define how a state s can satisfy a formula $\mathcal{R}_{\bowtie r}$ $(\varTheta)$ depending on the way the reward expression $\varTheta$ is formulated.

- $\mathcal{R}_{\bowtie r}(I^{=k})$ is true in state s if the expected state reward to be gained in the state entered at step k along the paths originating in s meets the bound $\bowtie r$.
- $\mathcal{R}_{\bowtie r}(C^{\leq k})$ is true in state s if, from state s, the expected reward *cumulated* after k steps meets the bound $\bowtie r$.
- $\mathcal{R}_{\bowtie r}(\diamond\varPhi)$ is true in state s if, from state s, the expected reward cumulated before a state satisfying $\varPhi$ is reached meets the bound $\bowtie r$.

The third construct can be used, for example, to state the global costs of the running systems, that is, until the execution reaches a *completion* state, usually modeled by an absorbing state because of its definitive nature.

A formal semantics for the reward fragment of R-PCTL can be found in [25]. Intuitively, the expected reward $\mathcal{R}(\varTheta)$ for all possible paths exiting a given state s and satisfying the pattern $\varTheta$ can be computed as the sum of the rewards for each path of those paths, weighted by the probability for that path to be taken. Even in case the set of paths originating from s is infinite, the resulting infinite series can be proved to converge [5]. Notice that the probability for a path to be taken is the joint probability of all its transitions to fire, which can be computed as the product of the probabilities associated with the transitions thanks to the Markov assumption[29]. We now need to define how the expected value X for the reward can be computed for a given path $\omega = s_0 s_1 s_2 \ldots$ of the D-MRM and for a given pattern:

$$X_{I=^k}(\omega) = \rho(s_k) \tag{2}$$

$$X_{C^{\leq k}}(\omega) = \begin{cases} 0 & \text{if } k = 0 \\ \sum_{i=0}^{k-1} \rho(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases} \tag{3}$$

$$X_{F\Phi}(\omega) = \begin{cases} 0 & \text{if } s_0 \models \Phi \\ \infty & \text{if } \forall i \ s_i \not\models \Phi \\ \sum_{i=0}^{min\{j|s_j\models\Phi\}-1} & \\ \quad \rho(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise} \end{cases} \tag{4}$$

In Section 3 we will show how the value of X_Θ can be computed with algebraic techniques taking into account the presence of both numeric values and variable parameters in the D-MRM model.

Exploiting rewards we are able to express more complex requirements which may consider for example costs or latencies. Let us recall our example of Figure 1 and let us imagine to deploy the system as follows. Let us imagine to have a separate machine for each server. In particular let us imagine the scenario in which we deploy the database and the file server on a Cloud infrastructure in which bandwidth and space are billed (e.g., Amazon Simple Storage Service[2]). In this setting we are now able to interpret the couple of numbers associated to each state in Figure 1. The first number indicates the average latency, in seconds, needed to process the request including the network latency. The second number indicates the average cost for each request for being processed by the state. For example the database server state has an average cost for each request equal to 0.07\$. Given these details we may express requirements such as:

- **R6**: *"The average cost for the system is less than* 0.03 \$ *for each request"*
- **R7**: *"The average response time for a given request is less than* $0.022s$*"*

These requirements can be translated into R-PCTL as shown in Table 2.

Table 2. Requirements translation in R-PCTL.

Req.	R-PCTL
R6	$R_{\leq 0.03}(true\,U\,7 \leq s \leq 9)$
R7	$R_{\leq 0.022}(true\,U\,7 \leq s \leq 9)$

3 Probabilistic Verification at Runtime: Existing Approaches

Standard verification techniques for PCTL properties over DTMCs are not suitable, in general, for runtime analysis because of the intrinsic time constraints

[2] http://aws.amazon.com/s3/

required by solvers. Some approaches have brought state-of-the-art probabilistic model-checkers at runtime [7], providing a suitable infrastructure for many applications. Nonetheless these approaches are not general enough for at least two reasons.

Notice that, the complexity of verification can be too high in case of large systems to make the analysis meet its time constraints [9,22]. Second, analysis procedures may be unsuitable for low power devices where the large number of operations required for mathematical iterative algorithms commonly used by model-checkers may result in excessive time and energy consumption.

Model-checking can be improved in many situations both in terms of analysis algorithms, e.g. by applying space-reduction techniques (e.g. [23,3]), and via the reuse of previous results, thus opening the way for incremental analysis [26].

Besides improving standard model-checking procedures, a different approach have recently gained relevance for runtime analysis. In its seminal work [11], Daws describes a procedure for parametric probabilistic model-checking of a subset of PCTL over DTMCs. This result trod an effective path for bringing probabilistic verification at runtime, by allowing to split the analysis process in two steps. The first consists in the parametric analysis of the model with respect to the desired property, whose result is a closed mathematical expression depending on the symbolic variables appearing in the model. This step is quite complex in terms of computational time, but it can be accomplished once for all at design-time, when time is usually not a strong constraint. At runtime all that is needed to obtain the actual analysis response is to replace the symbolic variables with the actual values provided from modeling, as soon as they are discovered. The evaluation of a mathematical expression, is in general a much simpler task than model-checking, and can be performed in a very short time even on low power devices, as we shown in [12].

The main focus of this section is on parametric probabilistic verification of PCTL properties over DTMCs. In Section 3.1 we will introduce the algorithm of Daws and the subsequent improvements and implementations. In Section 3.2 we present an alternative method for parametric analysis which overcomes the limitations of Daws' algorithm, covering the entire family of PCTL formulae with an improved performance.

3.1 State Elimination Algorithms

The first approach for parametric model-checking of DTMCs has been proposed in [11]. The main contribution of that seminal work concerned the synthesis of parametric closed formulae through a *state elimination algorithm*, analogous to the one used in automata theory to synthesize regular expressions from finite state automata [20].

More precisely, Daws' algorithm allows to compute a closed mathematical expression corresponding to the probability of reaching a set of target states in any number of steps. In terms of PCTL, this corresponds to computing $Pr(\mathit{true}\, U\ \phi)$, with the further constraint that ϕ can only be a boolean combination of atomic formulae, i.e. it has to be possible to identify the set of states in which ϕ holds at design time and this set is not going to change at runtime. We will refer to this family of reachability formulae as *flat*.

In the next section we will introduce Daws' algorithm for reachability analysis. Afterward, in Section 3.1, we will cover its extension for the analysis of a subset of rewards formulae.

Flat Reachability Analysis. The core idea of Daws' algorithm is to consider the probability values labeling DTMC transitions as letters of an alphabet. Under this interpretation the DTMC can be seen as a finite state automaton for which it can be synthesized a variant of the regular expressions by adapting the well known state elimination algorithm [20]. Such variants of the regular expressions are named stochastic regular expressions (SREs)[11] and can be evaluated to rational mathematical expressions. The construction of SREs corresponding to the evaluation of flat reachability formulae on a DTMC is addressed by the first part of this section.

Given a flat reachability formula *true* U ϕ and a DTMC D, it possible to identify the set of states T of D that satisfy ϕ. We will call these states *target* states.

In order to simplify the exposition, let us assume for now that all the target states are absorbing. We will later relax this assumption.

We also assume the model to be well-formed, meaning that all the states are reachable from the initial state s_0. We can also prune out all the states (and the corresponding transitions) from which it is not possible to reach any of the target states. The model we obtain may no longer be a DTMC, since the elimination of a subset of the transitions may lead to sub-stochastic states, i.e. the sum of the outgoing probabilities is lower than 1, nonetheless the reduced model preserves all the information needed for the computation of the reachability formulae (a proof of correctness can be found in [18]).

Daws' algorithm consist in eliminating all the states of the reduced model but the targets and the initial state. A state elimination step is described in Figure 2. When eliminating state s, the algorithm considers all the pairs (s_i, s_j) where s_i is a direct predecessor of s and s_j is a direct successor of s. When eliminating s, the transition probability from s_i to s_j is increased by a term representing the probability of reaching s_j from s_i through s. Such a term is, roughly, the sum of the probabilities of all the possible paths, that can be computed by iterating on the length k of a path:

$$\sum_{k=0}^{\infty} p_a p_c^k p_b = \frac{p_a p_b}{1 - p_c} \tag{5}$$

The state elimination terminates when the model is composed of the initial state s_0 directly connected to each of the target states. Each of these transitions will be labeled by an SRE representing the probability of reaching the specific target state. The value of $Pr(\textit{true}\ U\ \phi)$ is just the sum of all those SRE. As it can be guessed from Figure 2, an SRE is essentially a rational expression,

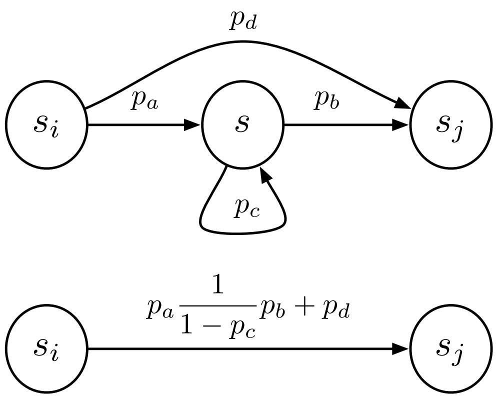


Fig. 2. SRE synthesis algorithm.

whose numerator and denominator are polynomials having as variable the labels of DTMC transitions.

In order to generalize the approach to deal with transient target states too, it suffices to pre-process the DTMC by making all the target states absorbing. Indeed, a formula $(\mathit{true}\ U\ \phi)$ is satisfied by a path as soon as it firstly reaches any of the states in which ϕ holds, hence its satisfiability is not affected if the states in which ϕ holds are made absorbing. Turning a transient state into absorbing could make other states unreachable from s_0; such unreachable states have to be pruned out in order to regain a well-formed model.

In [18], Daws' algorithm has been implemented in the tool PARAM. An effective improvement provided by PARAM to the original algorithm of [11] consists in replacing the transition labels corresponding to numeric transitions by their actual value after each state elimination. This allows to exploit arithmetic simplification of the intermediate SREs that can significantly speed-up both memory consumption and subsequent mathematical operations due to state eliminations, as shown in [18].

The result of executing PARAM is a closed rational expression having as variable only the symbolic parameters of the model, since numeric ones have been already evaluated by the tool. Such an expression is then evaluated at runtime.

In our example, requirement **R1** is formalized through a flat reachability property. Its parametric verification a design-time produces the following expression:

$$\begin{aligned} Pr(\mathit{true}\ U\ \ s=8) = {} & 1 - y - 0.7 \cdot w + 0.7 \cdot x \cdot w + 0.144375 \cdot z \cdot k + 0.7 \cdot y \cdot w \\ & + -0.7 \cdot y \cdot x \cdot w - 0.144375 \cdot k + 0.144375 \cdot y \cdot k \\ & -0.144375 \cdot y \cdot z \cdot k \end{aligned}$$

When the actual values of parameters x, y, w, z, and k become available at runtime, it would suffice to substitute them in the previous expression to obtain the probability of reaching state 8. If the obtained result is ≥ 0.999, then **R1** is satisfied. Otherwise it is not.

Cumulative Rewards Analysis A second major contribution of PARAM with respect to Daws' algorithm is its extension to deal with D-MRMs. Given as input a D-MRM D and a set of target states T, PARAM is able to compute the expected cumulative reward until a state in T is reached. The precise semantic of this measure has been provided in Equation (4).

The algorithm is again based on the state elimination procedure. Considering the pairs (s_i, s_j) of direct predecessors and direct successors of a state s, respectively, the goal is to obtain the transition reward $\iota(s_i, s_j)$ for the new transition from s_i to s_j after eliminating s. A step of this state elimination procedure is described in Figure 3, where a label p/r represents either the transition probability and the transition reward ι or the state name and its state reward ρ, respectively.

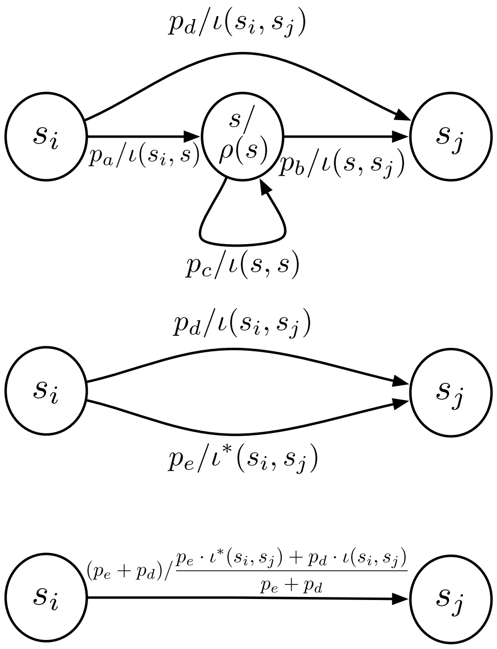


Fig. 3. State elimination for D-MRM.

The value of p_e is computed as for reachability analysis as $\frac{p_a p_b}{1-p_c}$, while the value of r_e can be computed as the sum of the reward accumulated over all the possible paths from s_i to s_j through s as (with respect to the path length k):

$$\begin{aligned} r_e &= \sum_{k=0}^{\infty}(p_a p_c^k p_b) \cdot (\iota(s_i, s) + \rho(s) + (\rho(s) + \iota(s,s)) \cdot k + \iota(s, s_j)) \\ &= \iota(s_i, s) + \rho(s) + \iota(s, s_j) + \frac{p_c}{1 - p_c}(\rho(s) + \iota(s,s)) \end{aligned} \tag{6}$$

The proof of correctness of the algorithms in this section can be found in [18].

As for reachability analysis, the resulting expected accumulated reward is again a rational expression with numerator and denominator being polynomials having as variables the symbolic parameters of the D-MRM, whether transition probabilities or rewards. The evaluation of such an expression at runtime requires just to replace the parameters with the numeric values coming from monitors, providing a far more efficient verification than model-checking.

In our example, **R6** requires the computation of an expected cumulated cost of a transaction. Its parametric analysis at design time produces the following expression:

$$\begin{aligned} X_{F(7 \leq s \leq 9)} = {} & 0.03810625 + 0.028 \cdot y \cdot x - 0.028 \cdot x - 0.03810625 \cdot y \\ & -0.01010625 \cdot z + 0.01010625 \cdot y \cdot z \end{aligned}$$

Design-Time Complexity. SRE can easily become very long and costly to manipulate. Indeed, analogously to regular expressions on finite state automata, the size of a SRE can grow as $n^{\Theta(logn)}$, where n is the number of states of the DTMC [16]. Such long expressions may take time to be manipulated at each state elimination step and may require a high memory consumption when the size of the model growths. The number of state elimination steps are in the order of $\Theta(n^3)$, as it can be easily proved [18], but the actual time each of them takes heavily depends on the complexity of the mathematical operation to be perfomed to combine SREs.

Though in the worst case $n^{\Theta(logn)}$ constitutes a complexity lower bound for computing SREs, in realistic software models most of the transitions can be assumed to be labeled by numeric values. Hence, by exploiting this assumption, instead of computing the full SRE taking transition labels as literals, PARAM intertwines the state elimination algorithm and the partial evaluation of numerical terms appearing in SREs. In other terms, at each step of the state elimination algorithm, the numeric labels are treated as numbers, thus allowing for the arithmetic simplification of intermediate results.

This induces a significant saving in the size of intermediate rational function representations, and hence an improvement in the actual computation time, as empirically shown in [18].

As a final remark, notice that the synthesis of the final rational expression may go through a large number of intermediate steps. In order to avoid any loss of accuracy, PARAM uses infinite precision rational numbers instead of double precision.

3.2 Linear Algebra Approaches

As shown in the previous section, PARAM is able to deal with reachability formulae and expected accumulated rewards, which are two subset of the properties that can be expressed in (R-)PCTL, though the most commonly used.

In this section we illustrate an approach, named WorkingMom (WM), able to deal with the entire (R-)PCTL to obtain a set of parametric closed formula through the use of linear algebraic algorithms. We will firstly show how to compute flat reachability formulae and then generalize the approach to cover the entire PCTL. Afterwards, we will present the algorithms to deal with the reward fragment of R-PCTL.

Flat Reachability Analysis. We start by focusing on flat reachability formulae for absorbing states. Recalling the structure of the transition matrix for an absorbing DTMC given in Equation (1), the matrix $I-Q$ (where I is the identity matrix of the same size of Q) has an inverse N and $N = I+Q+Q^2+Q^3+\cdots = \sum_{i=0}^{\infty} Q^i$ [29]. Recall from Section 2.1 that an entry q_{ij} of the matrix Q represents the probability of moving from the transient state s_i to the transient state s_j in exactly one time step. The entry n_{ij} of N represents the number of times the Markov process is expected to visit the transient state s_j before being absorbed, given that it started from state s_i. Notice that a Markov process is considered absorbed when it reaches any of the absorbing states. Notice that $Q^n \to 0$ when $n \to \infty$ (as discussed in Section 2.1), thus after enough time the process will always eventually be absorbed, no matter which state it started in.

Every time the process accesses a transient state s_i, it has a probability of being absorbed in the next time step in the absorbing state s_j given by the entry r_{ij} of the matrix R. Generalizing to all the pairs (s_i, s_j) where s_i is transient and s_j is absorbing, we can get the absorbing distribution B of the DTMC as:

$$B = N \times R$$

An entry b_{ij} of the matrix B represents the probability for the process of being eventually absorbed in s_j (in any number of states), given that it started from s_i. B is by construction a $t \times r$ matrix, where t is the number of transient states and r the number of absorbing ones.

Given a DTMC D and a set T of target absorbing states, the probability of reaching T from the initial state s_0 can be computed as:

$$Pr(true\ U\ T) = \sum_{s_j \in T} b_{0j} \tag{7}$$

The goal of design-time pre-computation is to compute the value of Equation (7).

Depending on the size of the system and the availability of a parallel execution environment the computation of the matrix B can be performed in different ways. In [12] we introduced a computational approach entirely based on matrix operations that can be straightforwardly suitable for parallelization. In this paper we will instead focus on how to efficiently compute matrix B in a sequential environment.

An entry b_{ij} can be computed, by definition of matrix product, as:

$$b_{ij} = \sum_{k=0..t-1} n_{ik} \cdot r_{kj} \tag{8}$$

Entries r_{ij} are readily available from matrix R. Entries n_{ik} belongs instead to the i-th row of matrix N, that is the inverse of $I - Q$.

By recalling the definition of inverse of a generic square matrix A, we know that $A \cdot A^{-1} = I$. Thus, if we are interested in computing the i-th column of the matrix A^{-1} we can simply solve the following linear system of equations:

$$A \cdot v = e_i \tag{9}$$

where e_i is the i-th column of the identity matrix, i.e. a column vector having all zero elements but for the i-th that is 1, and v is the unknown vector corresponding to the i-th column of A^{-1}. Since in Equation (8) we are required to know the entries of the i-th row of the matrix $N = (I - Q)^{-1}$, we can exploit a property of the transpose of invertible matrices, namely $(A^{-1})^T = (A^T)^{-1}$), to compute those entries.

Indeed, we are interested to the i-th row of $(I - Q)^{-1}$, which is equal to the i-th column of $((I - Q)^{-1})^T)$, which is in turn equal to the i-th column of $((I - Q)^T)^{-1}$, by the just mentioned property.

The problem of calculating the a row of the matrix N and, through (8), of B can be reduced to the solution of a linear system of equations. This solution may take a long time to be performed by using out of the shelf algorithms. Though the peculiarities of many DTMC classes, such as the models derived from software artifacts, can be effectively exploited to improve the design time efficiency, as it will be later discussed in Section 3.2.

The solution of (8) leads again to the generation of a closed rational expression, equivalent to the one computed by means of PARAM. This expression can then be brought at runtime for efficient evaluation as soon as monitors provide the actual values for symbolic parameters.

Analogously to Section 3.1, in order to generalize the procedure to the reachability of transient states it is sufficient to pre-process the model by making the target transient states absorbing. As already said, this operation may make some of the states of the DTMC unreachable from s_0. The unreachable states have to be pruned to obtain a well-formed model.

Extending to Entire PCTL. Flat reachability is the most widely used type of PCTL properties [17]. Nonetheless there are relevant requirements that cannot be easily expressed in terms of flat reachability formulae.

In this section we will incrementally show how to handle the entire PCTL by means of the WM approach. We will start by extending the reachability approach to the case of generic flat until formulae $\mathcal{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$, where ϕ_1 is no longer constrained to be equal to *true*. Afterward we will present algorithms to verify the bounded operators X and $U^{\leq t}$. Finally we will relax the constraint for the inner state formulae to be flat and allow the nesting of temporal operators, thus covering the entire PCTL.

Flat Until Formulae. The core idea for analyzing generic flat until formulae is to reduce the problem to the analysis of equivalent reachability formulae, and then apply the solution procedures already seen.

Starting from a DTMC D and a flat until formula $\mathcal{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$, we will construct a new DTMC $\bar{D}$ and a flat reachability formula upon $\bar{D}$ equivalent to the desired property of D. In order to construct $\bar{D}$ the following procedure has to be applied on D:

1. Add two absorbing states s_{goal} and s_{stop}
2. For all the states where ϕ_2 holds, remove all the outgoing transitions and put a single one (with probability 1) toward s_{goal}
3. For all the states where $\neg(\phi_1 \vee \phi_2)$ holds, remove all the outgoing transitions and put a single one toward s_{stop}.

Computing on $\bar{D}$ the flat reachability property $\mathcal{P}_{\bowtie p}$ (true $U\ s_{\text{goal}}$) provides the same result as computing the flat until probability of $\mathcal{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$.

The rationale behind the previous procedure is that a path satisfying $(\phi_1\ U\ \phi_2)$ cannot pass from any state where neither ϕ_1 nor ϕ_2 hold (point 2) and has to eventually reach a state where ϕ_2 holds (point 1). At this point it is possible to apply the same mathematical machinery previously introduced for flat reachability of absorbing states, namely the solution of Equation (8) for the entry $b_{0\ s_{\text{goal}}}$.

As a final remark, notice that flat reachability formulae are special cases of flat until ones. They have been presented separately for the sake of simplifying the exposition.

Flat Next and Bounded Until. Let us focus now on the parametric analysis of Next and Bounded Until flat formulae.

The set of paths to be considered in order to estimate the probability of a path formula $X\ \phi$ in a state s_i is composed by all the 1-step long paths originating in s_i. Under the hypothesis of flat formulae, the truth of ϕ can be computed once for all at design time. As we stated in Section 2.1, the transition matrix P contains the probability of moving from a state to another in a single step. Hence, to compute the probability of reaching, from a state s_i, a state where ϕ holds in 1 step, the following procedure is in place:

$$Pr(X\ \phi_1) = \sum_{s_j \models \phi_1} p_{ij} \tag{10}$$

For example, applying (10) in state s_3 to verify the requirement **R5** of our example leads, as it should be easy to guess, to $1 - x$.

A similar procedure applies to the case bounded flat until. Indeed, each path originating in s_i and satisfying $\phi_1 U^{\leq t} \phi_2$, at a certain step $k \leq t$ will reach a state s_j where ϕ_2 holds, and for all the previous steps ϕ_1 has to hold. If we exploit the same construction used in the case of flat until formulae to construct the modified DTMC $\bar{D}$, each of these paths corresponds to a path in $\bar{D}$ that exactly at time step $k+1$ reaches the state s_{goal}, by construction. Being s_{goal} an absorbing state it is non going to be left by the path. Hence, we can conclude that any path of D satisfying $\phi_1 U^{\leq t} \phi_2$ corresponds to a path in $\bar{D}$ being at time $t+1$ in state s_{goal}.

The probability distribution of the states reached by a DTMC after exactly $(t+1)$ time steps can be computed by elevating the transition matrix P to the $t+1$-th power:

$$Pr(\phi_1 U^{\leq t} \phi_2) = (P^{t+1})_{s_0 s_{\text{goal}}} \tag{11}$$

Nested Formulae. We have so far restricted the analysis of PCTL formulae to what we called the flat fragment, that is, the set of formulae where the arguments of a path operator are boolean combinations of atomic propositions only. The peculiarity of flat formulae is that it is always possible at design time to identify the states where a state formula ϕ holds, and thus generate a parametric expression by means of the procedures previously exposed.

In the case of *nested* formulae, that is formulae $\mathcal{P}_{\bowtie p}\ (\Psi)$ where at least one sub-formula of ψ is again a path formula, some information needed to compute the desired parametric expression may only become available at runtime. For example, the set of states satisfying **R1** will be known only at runtime, because it depends on the actual values assigned to the model parameters. If for example such a state formula would appear as the right-hand operand of an until operator, it would not be possible to apply at design time the procedures exposed so far, since it would not be possible to identify the target states. Indeed, to evaluate a formula with nested $\mathcal{P}_{\bowtie p}\ (\cdot)$ operators, so far we needed to know in which states its sub-formulae are satisfied, and this, in general, depends on the value of the model parameters. The same consideration can be applied recursively to sub-formulae of a sub-formula, until we reach a flat one that can be directly analyzed.

To deal with this issue we want to delay at runtime the evaluation of a nested formula, when all the knowledge concerning its sub-formulae has been gathered, without loosing the benefits of parametric verification.

Let us focus on until formulae. The solution previously provided is based on the construction of the modified DTMC $\bar{D}$. Such a construction requires to disconnect certain states from their successors and to connect them to either s_{goal} or s_{stop}. Then, for what has been previously explained, the resulting parametric expression would be the entry $b_{s_0 s_{\text{goal}}}$ of the matrix B computed as in (8) on $\bar{D}$. In order to delay at runtime the decision about the connection of a state to s_{goal} or to s_{stop}, all is needed is the addition of three more parameters per state. The first will be a coefficient m_i that multiplies all the elements p_{ij} of D. The second and the third are, respectively, two terms $a_{i\text{goal}}$ and $a_{i\text{stop}}$ to be put in correspondence of the entries $p_{s_i s_{\text{goal}}}$ and $p_{s_i s_{\text{stop}}}$ of the matrix P of $\bar{D}$. The three additional parameters can assume values 0 or 1, and their intuitive purpose is the following: assigning 0 to a parameter m_i disconnects state s_i from all its successors; assigning 1 to either $a_{i\text{goal}}$ or $a_{i\text{stop}}$ connects state s_i to state s_{goal} or s_{stop}, respectively.

Computing $b_{s_0 s_{\text{goal}}}$ at design time will lead to a parametric expression having as variables both the model parameters and the additional parameters m_i, $a_{i\text{goal}}$, and $a_{i\text{stop}}$ for each state s_i. At runtime, when information about the sub-formulae of a nested formula becomes available, the value of the additional parameters can be set in order to adapt the expression to reflect the convenient transformation of $\bar{D}$. Applying this procedure recursively on nested formulae allows to keep the benefits of parametric analysis, though it would require at most as many evaluations as the nesting depth of the formulae. Assuming most of the nested formulae to have just a few nesting levels, the impact on runtime complexity would still be limited. Another drawback in parametric analysis of nested formulae is that the resulting mathematical expressions could be longer than in the case of flat formulae due to the presence of more parameters, but the evaluation time is still not comparable with the execution of a model-checking routine for a system large enough.

Finally, the computation of next and bounded until nested formulae follows the same principle described for until ones, and they have to be computed on the model instrumented with the additional parameters m_i, $a_{i\text{goal}}$, and $a_{i\text{stop}}$. The adaptation of the mathematical procedure for the Next operator is a trivial exercise.

Reward Analysis. Equations (2), (3), and (4) of Section 2.4 formalize the semantics of the three specification patterns for reward formulae defined for R-PCTL. In this section we will provide mathematical algorithms for the analysis of each of them.

The following mathematical procedures are based on the notion of expected reward along a set of paths originating from a state s_i. In Section 2.4 this value has been intuitively defined as the sum of the rewards cumulated along each of those paths, weighted by the probability for that path to be taken. Since such a sum may contain infinite terms and could be unfeasible to compute directly, we need a different procedure more suitable for an efficient mathematical solution. Exploiting the Markov property and the linearity of the expected value [29], the computation of the expected reward for a (non empty) path originating in s_i can be computed as the sum of the state reward $\rho(s_i)$ gained in state s_i and the expected reward to be gained in each of the possible successors of s_i, weighted by the probability of moving toward it. Applying this observation to all the states S of a D-MRM leads to the following linear system of equations:

$$r_i = \rho(s_i) + \sum_{s_j \in S} p_{ij} \cdot (\iota(s_i, s_j) + r_j) \tag{12}$$

where r_i is the expected reward over all the paths originating in s_i

In order to simplify the exposition, we will refer in this section only to flat R-PCTL formulae, meaning that in path formulae $\diamond\phi$, ϕ may not contain any of the occurrence of the modal operators $\mathcal{P}_{\bowtie p}(\cdot)$ and $\mathcal{R}_{\bowtie r}(\cdot)$. The extension to the nested fragment of R-PCTL can be achieved by instrumenting the D-MRM with additional parameters as it has been done previously for nested PCTL formulae. As further simplifying assumption, in this Section we will focus on state rewards only. Transition rewards can always be mapped into state rewards of a modified D-MRM automatically.

Let us start with the parametric analysis of formulae $\mathcal{R}_{\bowtie r}(\diamond\phi)$. Recalling (4) and (12), we can define the computation of the expected cumulated rewards over all the paths satisfying $\diamond\phi$ and originating in a state s_i as the solution of the following linear system of equations:

$$r_i = \begin{cases} 0 \text{ if } s_i \models \Phi \\ \infty \text{ if } s_i \text{ is absorbing and } s_i \not\models \Phi \\ \rho(s_i) + \sum_{s_j \in S} p_{ij} \cdot r_j \;\; \text{otherwise} \end{cases} \tag{13}$$

The rational behind (13) is intuitive: a state s_i satisfying ϕ marks the satisfaction of the path formula $\diamond\phi$ and thus the end of the reward accumulation, on the other hand, an absorbing state that does not satisfy ϕ marks a path that will never satisfy $\diamond\phi$ and thus contribute to the accumulation of rewards as an infinite cost, as from the definition in (4).

Notice that the solution of (13) leads to a polynomial expression having as variables the model parameters, whether they label transition probabilities or state rewards. For example, the parametric verification of requirements **R7** leads to the following expression (notice that in this case we are considering as state reward the average execution time):

$$\begin{aligned} X_{F(7\leq s\leq 9)} = {} & 0.21734375 + 0.084 \cdot y \cdot x - 0.084 \cdot x - 0.21734375 \cdot y \\ & -0.02165625 \cdot z + 0.02165625 \cdot y \cdot z \end{aligned}$$

Concerning formulae $\mathcal{R}_{\bowtie r}(I^{=k})$, from (2) it can be computed as the sum of the rewards of every state reached in exactly k time steps, weighted by the probability of reaching it. Recall that the probability of reaching a state s_j from a state s_i in exactly k time steps is the entry $(p^k)_{ij}$ of the matrix P^k. Let us define the column vector $\bar{\rho} = [\rho(s_0), \rho(s_1), \rho(s_2), \dots]$. The expected reward $X^{=k}$ can be computed for all the paths originating from a state s_i by the following equation:

$$X_{I^{=k}} = P^k \cdot \bar{\rho}\Big|_i \tag{14}$$

where $|_i$ indicates the i-th element of the resulting vector.

Finally, formulae $\mathcal{R}_{\bowtie r}(C^{\leq k})$ require to compute the cumulated reward along all possible paths up to the k-th step. For the previous consideration, the expected reward gained at the j-th step is exactly $P^k \cdot \bar{\rho}$. Thus, to compute the cumulated reward up to the k-th step with $k \geq 1$ it is possible to apply the following equation:

$$X_{C^{\leq k}} = \sum_{i=0}^{k-1} P^i \cdot \bar{\rho} \Big\|_0 \tag{15}$$

When $k = 0$, $X_{C^{\leq k}} = 0$ by definition (3).

This section concludes the exposition of the mathematical machinery used within the WM approach. Most of the mathematical procedures exposed so far relies on the ability to efficiently and accurately solve a linear system of equations. In the next section we will briefly sketch the basics of the solution strategy currently used in the WM approach.

Design-Time Complexity. Solving linear systems of equations is a well studied mathematical problem, even though most of the computational approaches concern numerical solution and cannot deal with symbolic parameters [28]. The most popular algorithms to solve linear equation systems embedded in probabilistic model-checkers are iterative ones [30,28], which can efficiently solve even large systems with the desired precision in the final result and without requiring a large amount of memory.

In the WM approach it is no possible to adopt the same strategy because iterative methods do not deal conveniently with symbolic parameters. Indeed, the presence of unknown parameters makes hard to assess the convergence of the iterative algorithm. For this reason we adopted a *direct* method to solve the system. Direct methods have the additional benefit of not loosing precision in the results, and both parallel and sequential algorithms have been provided. More specifically we are interested in direct methods for the solution of sparse linear systems [10] because a Markov model for a software system is likely to have only a few non-zero entries for each row of the matrix Q, since a component or a task are usually designed to directly interact with only a few counterparts.

Sparsity of the linear system can be exploited to obtain a faster computation. Since [13], we implemented a solver based on structured Gaussian elimination and Markowitz pivoting [10]. Structured Gaussian elimination is a variation of the widely used method to triangularize linear systems which allows to reduce the solution of a large sparse equation system to the solution of a small dense one. This collapse can significantly reduce the size of the system to be actually solved. A core element of structured Gaussian elimination is the strategy used to select the order in which elements of the original system will be eliminated. In fact, each elimination step may reduce the sparsity of the obtained system, reducing in turn the global effectiveness of the method. This problem is known as *fill-in*. In order to reduce the fill-in during the elimination steps we adopted Markovitz pivoting as a selection strategy of the next element to be eliminated. Other strategies can be more suitable for specific cases but their discussion as well as mathematical details concerning structured Gaussian elimination and Markovitz pivoting are beyond the scope of this paper. The interested reader may refer for example to [10].

Finally, in order to avoid any loss of accuracy during intermediate computation steps, the WM solver uses infinite precision rational numbers for all the numeric values appearing in the models. All the mathematical procedures for the WM approach have been implemented in Maple 15[3].

4 Empirical Evaluation

In this section we provide an empirical evaluation of the tools presented in Section 3. For the verification we focus on the property $\mathcal{R}_{\bowtie r}\ (F\phi)$, where ϕ identifies the unique absorbing target state defined in all the test cases. We chose this property since it embeds a reachability formula that is both the most widely used in practical verification and the most complex to compute for the two tools. We are interested in evaluating the execution time of just the design time phase. The runtime verification, being just the evaluation of polynomial forms, takes a very short time even for very large parametric formulae, as discussed in [12].

We will provide a first comparative study of the two approaches with respect to two dimensions of the problem, namely the number of states and the number of symbolic parameters. Though this cannot be considered a complete comparison of the approaches, it provides a glimpse of how the their current implementations scale with respect to the two dimensions investigated and gives to the reader an insight about the actual computation time needed for parametric analysis of D-MRM models.

Beside PARAM and WM we added a graph from a modified WM where the linear system of equation is solved by means of the built-in solver of Maple 15. This would provide an evidence of the effectiveness of the chosen solution strategy for the actual WM.

All the models used for the tests are well-formed and generated randomly. Since we are interested in comparing the efficiency of verification algorithms, we disabled the pre-processing procedure implementing state-reduction algorithms for D-MRM that are enabled by default in PARAM. The same pre-processing could be implemented also for the WM, but is out of of the scope of this paper.

The execution environment is a Dual Intel(R) Xeon(R) CPU E5530 @ 2.40 GHz with 8Gb of ram, equipped with GNU Linux Ubuntu server 11.04 64bit. All the tests considered in this section did not overrun the available memory.

Due to the high variability in the actual execution time, we reported the average execution time with a thick black line and the maximum measured execution time in a dashed thin line.

Figure 4 reports the execution time of the two tools with respect to the number of states. All the samples have exactly 5 outgoing transitions from each transient state. There are 5 parametric transitions and 2 parametric rewards for a total of 7 symbolic parameters. The sample set is composed by 50 samples.

As shown in Figure 4(a), the execution time slightly grows with respect to the number of states for PARAM. Nonetheless there is a strong variability in the

[3] `http://www.maplesoft.com`

average execution time due to the impact on the solver of the specific topology of each case. This factor will affect most of the tests proposed in this section and need further investigations to conveniently characterize input model with respect to the topology of their D-MRM. A main difference between PARAM and the approaches based on linear algebra is the order of magnitude of the execution times, being in the first case up to 10^4 time higher. There are instead no significant differences between the Maple built-in solver and the WM.

Figure 5 shows the execution time of the three solver when the number of parameters changes. All the models have exactly 100 states, with 5 outgoing transitions per state. The parameters are distributed between transitions probability and rewards, without duplication of the same symbol.

We limited the number of parameters to 10 because of the long execution time required by PARAM, as shown in Figure 5(a). In this figure, it is clear that the execution time grows sharply with the number of parameters, taking more than 2h to process models with just 10 symbolic parameters. Maple built-in solver provides a quite reasonable performance, taking no more the 0.2s in our tests, while the WM is slightly faster than this.

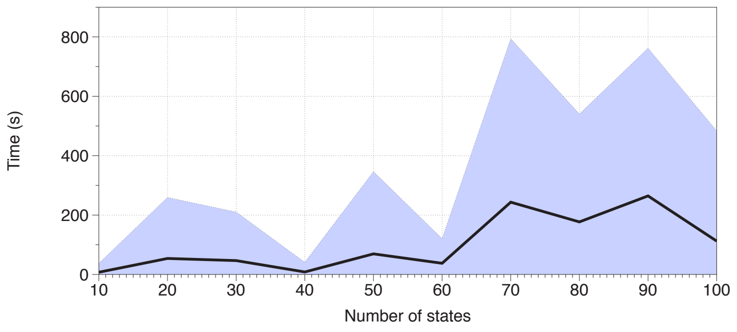
When the number of parameters growths up to 45, the benefits of the WM solution strategy become more visible (Fig. 6). Indeed, the built-in solver of Maple reaches a maximum execution time of about 3h, while the WM took no more than 7 minutes in the worst case. This last analysis has been performed on 200 randomly generated test cases, 50 per observation point.

Concluding, besides a global glimpse of what could be the actual execution time of state of the art tools for parametric verification of Markov models, the tests in this section show that the number of states and the number of parameters do not satisfactorily characterize the performance of the solver. Indeed there is a relevant distance from the maximum and the average execution time as a sign of the high variance in the measures. This suggests that specific topologies my affect, positively or negatively, the performance of the solver so far implemented, and need further investigation.

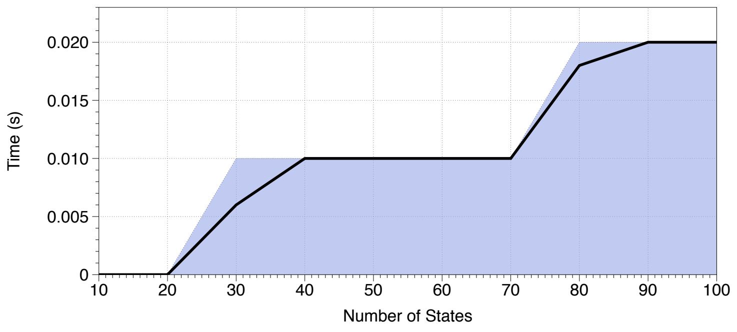
5 Conclusions and Future Work

Evolving systems require efficient verification procedure in order to timely reveal violations of their requirements. Many quantitative attribute related to the quality of service provided by the system heavily depends on environmental factors, such as the usage scenario and the interactions with external components. Those environmental factors are often out from our control and subject to unpredictable changes. A probabilistic framework could be a convenient mean to deal with this uncertainty and the availability of probabilistic model-checkers allows one to formalize and verify quantitative requirements in a fully automatic way.

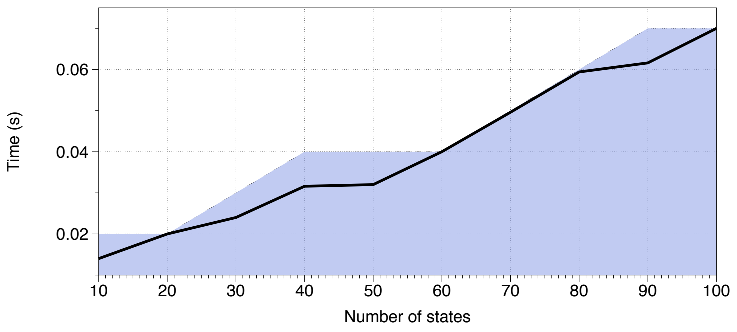
Nonetheless, re-running a model-checker after any detected change may hamper the verification performance, making the system unresponsive or unable to identify the problem on time. Parametric model-checking shown to be an effec-



(a) PARAM



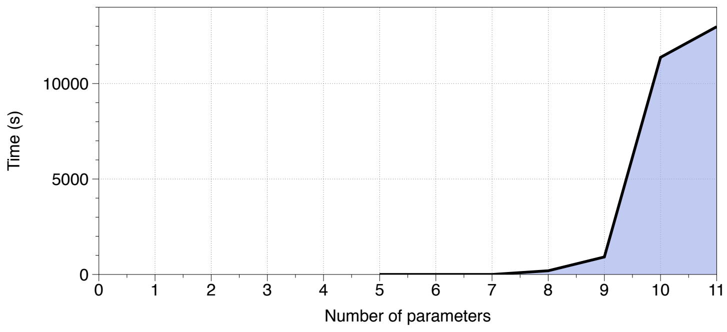
(b) Maple built-in solver



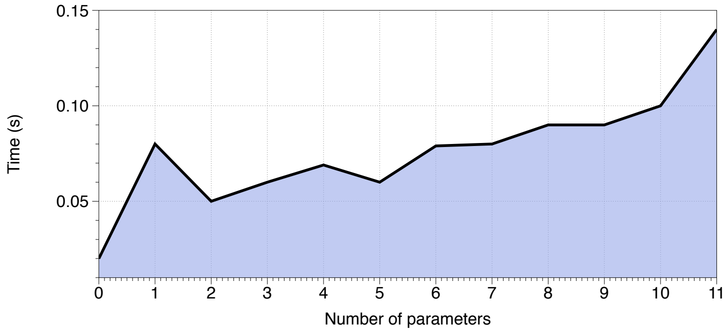
(c) WorkingMom

Fig. 4. Execution time vs number of states.

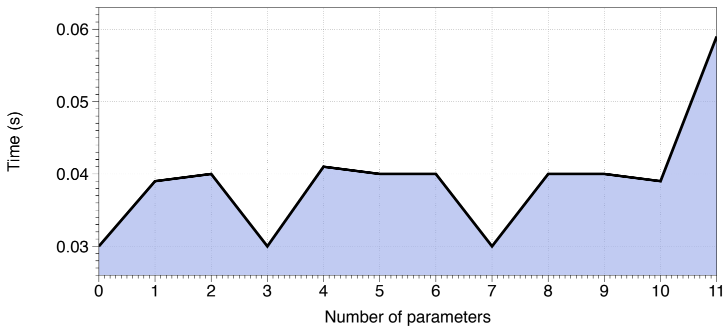
tive replacement for evolvable systems since it allows to partially evaluate the requirements at design time producing closed mathematical formulae quickly evaluable at runtime. The main burden of parametric model checking consists in design time computation that can be quite expensive in terms of computational time. Although the time is not supposed to be a too strict issue during design,



(a) PARAM



(b) Maple built-in solver

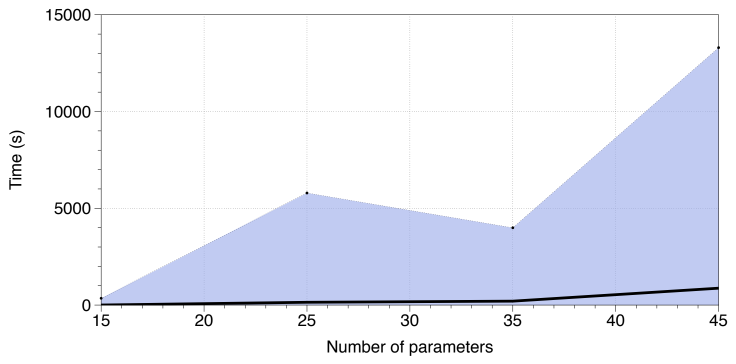


(c) WorkingMom

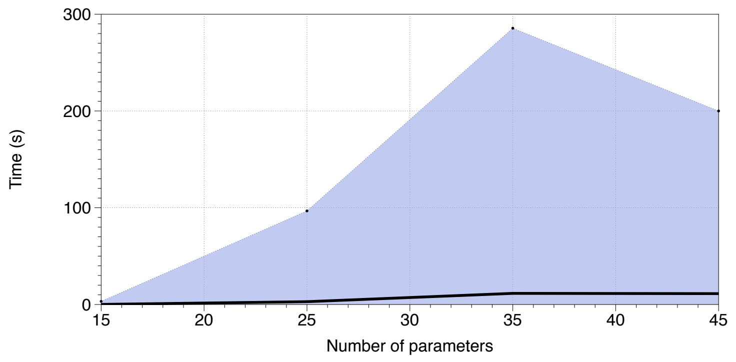
Fig. 5. Execution time vs number of parameters.

the availability of efficient tools could speed up the entire process and provide a better interaction with the system designers.

The state of the art parametric verifiers provide reasonable performances for design-time computation, though their execution time strongly depends on the



(a) Maple built-in solver



(b) WorkingMom

Fig. 6. Execution time vs number of parameters.

topology of each input model. This dependency has to be further investigated in order to select for each input the most efficient methodology.

Future steps in probabilistic parametric verification should focus on scalability issues that may arise in case of large models, as well as on supporting more complex models and properties, such as continuous time Markov chains that are widely used for software performance analysis. A further limitation of current tools is that they do not allow changes in the structure of the model that are not expressible as an assignment to its parameters. Overcoming this limitation could open the way to a significantly broader application in the field of self-adaptive systems.

Acknowledgments

This research has been funded by the EU, Programme IDEAS-ERC, Project 227977-SMScom and FP7-PEOPLE-2011-IEF, Project 302648-RunMore.

References

1. S. Andova, H. Hermanns, and J.P. Katoen. Discrete-time rewards model-checked. *Formal Modeling and Analysis of Timed Systems*, pages 88–104, 2004.
2. A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *Computer Aided Verification*, pages 155–165. Springer, 1995.
3. Christel Baier, Pedro D'Argenio, and Marcus Groesser. Partial order reduction for probabilistic branching time. *Electronic Notes in Theoretical Computer Science*, 153(2):97–116, 2006.
4. Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29:524–541, 2003.
5. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.
6. L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.
7. R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimisation in service-based systems. *Software Engineering, IEEE Transactions on*, (99):1–1, 2010.
8. R. C. Cheung. A user-oriented software reliability model. *IEEE Trans. Softw. Eng.*, 6(2):118–125, 1980.
9. C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4), 1995.
10. T.A. Davis. *Direct methods for sparse linear systems*, volume 2. Society for Industrial Mathematics, 2006.
11. C. Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proc. of ICTAC 2004*, volume 3407 of *LNCS*, pages 280–294. Springer, 2005.
12. A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *33 International Conference on Software Engineering (ICSE11)*, accepted for publication.
13. Antonio Filieri and Carlo Ghezzi. Further steps towards efficient runtime verification: Handling probabilistic cost models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, Zurich, 2012.
14. Katerina Goseva-Popstojanova and Kishor S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179 – 204, 2001.
15. C.M. Grinstead and J.L. Snell. *Introduction to Probability.* Amer Mathematical Society, 1997.
16. Hermann Gruber and Jan Johannsen. Optimal lower bounds on regular expression size using communication complexity. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 273–286. Springer, 2008.
17. Lars Grunske. Specification patterns for probabilistic quality properties. In *ICSE*, pages 31–40, 2008.
18. E. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric markov models. *Model Checking Software*, pages 88–106, 2009.
19. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

20. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-wesley, 2007.
21. A. Immonen and E. Niemel
"a. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.
22. David Jansen, Joost-Pieter Katoen, Marcel Oldenkamp, Marille Stoelinga, and Ivan Zapreev. How fast and fat is your probabilistic model checker? an experimental performance comparison. LNCS, pages 69–85. Springer, 2008.
23. Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, and David Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 87–101. Springer, 2007.
24. J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE'07*, pages 259–268. IEEE, 2007.
25. Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *Formal Methods for Performance Evaluation*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.
26. Marta Z. Kwiatkowska, David Parker, and Hongyang Qu. Incremental quantitative verification for markov decision processes. In *DSN*, pages 359–370, 2011.
27. Caporuscio M., Funaro M., and Ghezzi C. Architectural issues of adaptive pervasive systems. In *Graph Transformations and Model-Driven Engineering*, pages 492–511, 2010.
28. A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37. Springer Verlag, 2007.
29. S.M. Ross. *Stochastic Processes.* Wiley New York, 1996.
30. Y. Saad. *Iterative methods for sparse linear systems.* Society for Industrial Mathematics, 2003.