Run-Time Efficient Probabilistic Model Checking

Antonio Filieri
Politecnico di Milano
DeepSE Group at DEI
Piazza L. da Vinci 32, 20133
Milano, Italy
filieri@elet.polimi.it

Carlo Ghezzi
Politecnico di Milano
DeepSE Group at DEI
Piazza L. da Vinci 32, 20133
Milano, Italy
ghezzi@elet.polimi.it

Giordano Tamburrelli
Politecnico di Milano
DeepSE Group at DEI
Piazza L. da Vinci 32, 20133
Milano, Italy
tamburrelli@elet.polimi.it

ABSTRACT

Unpredictable changes continuously affect software systems and may have a severe impact on their quality of service, potentially jeopardizing the system's ability to meet the desired requirements. Changes may occur in critical components of the system, clients' operational profiles, requirements, or deployment environments.

The adoption of software models and model checking techniques at run time may 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 simply 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.

This paper precisely addresses this issue and focuses on reliability models, given in terms of Discrete Time Markov Chains, and probabilistic model checking. It develops a mathematical framework for run-time probabilistic model checking that, given a reliability model and a set of requirements, statically generates a set of expressions, which can be efficiently used at run-time to verify system requirements. An experimental comparison of our approach with existing probabilistic model checkers shows its practical applicability in run-time verification.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking, Reliability*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques, Performance attributes*

General Terms

Probabilistic Model Checking, Reliability

Keywords

Discrete Time Markov Chains, Run-Time Model Checking

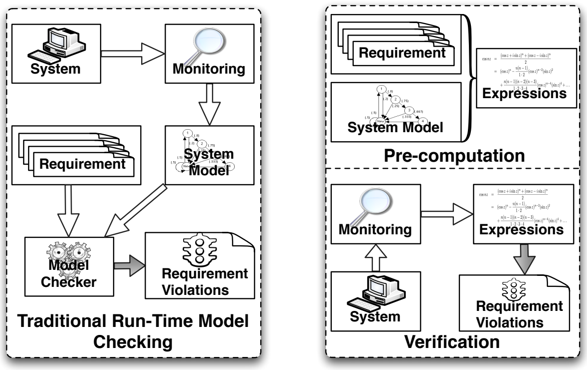
1. INTRODUCTION

Often software systems are designed, developed and implemented to operate in a completely known and immutable environment with stable requirements and unvaried operational profiles. In this setting, each change is unexpected and may jeopardize the ability of the system to meet its requirements. Whenever a software has to be changed, a complete maintenance lifecycle–design, development, and deployment of a new version of the system–has to be planned. In this scenario changes are considered harmful and lead to costly maintenance activities and unsatisfactory time-to-market.

Increasingly, however, changes occur very frequently and constitute one of the dominant factors of current software systems. Today's software is often built through composition of components operated by independent organizations (e.g., Web services integrated in a larger system), which may evolve unpredictably; clients' operational profiles and deployment environments may also change over time. As a consequence, software engineers are increasingly required to design software as an *adaptive system*, which automatically detects and reacts to changes.

Many current research proposals describe methodologies and techniques to design adaptive systems. In this paper, we focus on software systems that try to adapt themselves to keep satisfying *reliability requirements* in the presence of changes. The most promising solutions build on top of two complementary techniques: *monitoring* and *models* (e.g., [12, 28, 29]). The former aims at interpreting data extracted at run time from instances of the system. The data collected by the monitor are analyzed to continuously update the parameters of the model (e.g., failure probability of an external service), to keep the model consistent over time with the changing behavior of the environment. The updated model may be analyzed by *model checking* [2, 7] tools, which verify the compliance between the current behavior of the system and the desired requirements. Because of our focus on reliability, the models we consider here are *Discrete Time Markov Chains* (DTMCs) [14].

Unfortunately, traditional model checking techniques and tools are conceived for design-time use and can hardly satisfy the execution time constraints normally imposed by run-time analyses because of the well known problem of state explosion, which occurs in analyzing the model. In particular, the use of model checking tools at run time leads to unsatisfactory execution time. Indeed, traditional model checking techniques take as input a model of the system and a property expressed in an appropriate formalism (i.e., the requirement) and verify if the former is compliant with respect to



(a) Use of Conventional Model Checker. (b) The Proposed Approach.

Figure 1: Run-Time Model Checking Techniques

the latter (i.e., the requirement is met by the model). As previously introduced, the monitoring continuously updates the system model at run time and the model checking process is periodically activated. This run-time procedure is computationally expensive and requires the exhaustive exploration of the model's state space—which may be very large—and the analysis of the property—which may be arbitrarily complex. Figure 1(a) summarizes such approach. The details concerning the complexity of traditional probabilistic model checking can be found in [2, 10, 23].

This paper focuses on efficiently evaluating the satisfaction of reliability requirements at run time. The key concept of the proposed solution relies on separating the model-checking activity in two distinct steps, executed at design time and run time, respectively. We refer to the design-time step as *pre-computation* and to the run-time step as *verification*. The pre-computation step takes as input: (1) the model of the system as a DTMC, (2) a set of *transition variables*, and (3) the reliability requirements of the system. The transition variables are the parameters of the model whose value becomes known only at run time, and may change over time. For example, a transition may connect a state modeling a service invocation to a failure state, and the transition variable is a literal representing the changing value of the service's failure rate. The output produced by the pre-computation step is a set of symbolic expressions which represent satisfaction of the requirements. The verification step performed at run time simply evaluates the formulae by replacing the variables with the real values gathered by monitoring the system. In the example, the monitor would yield the real value of the failure rate of the service and the formulae representing the requirements would evaluate to either true or false (in case of a violation). Figure 1(b) describes the two steps involved in the proposed approach.

The main advantage of our approach relies on shifting the cost of model analysis at design time. The (computationally expensive) design-time transformation of reliability properties into symbolic formulae reduces run-time model checking to substituting variables with values and evaluating the expression, which is computationally inexpensive and does not require model exploration. The rationale behind the approach is that we are willing to pay for an expensive transformation step at design time if run-time analysis becomes efficient, and amenable to on-line processing. We measured the speed up obtained by our run-time model checking approach with respect to existing probabilistic model checkers: PRISM [18] and MRMC [21] pointing out advantages and threats to validity of both the approaches. As shown in Section 4, our method outperforms existing probabilistic model checkers under the assumption that potential changes can be anticipated and the number of variable transitions is small. In the extreme case, one may of course assume all transitions to be variable, but this would make our approach impractical.

The rest of the paper is organized as follows: Section 2 provides the necessary background. Section 3 describes the proposed approach. Section 4 illustrates the simulations performed through a tool we implemented and reports the experimental results we obtained. Section 5 discusses related work. Section 6 concludes the paper describing some current limitations of our approach and future work.

2. GROUNDING THE PROBLEM

We assume the system under development to be modeled as a Discrete Time Markov Chain (DTMC). DTMCs a widely accepted formalism to model reliability of component (service)-based systems. In particular, they proved to be useful for an early assessment or prediction of reliability [19]. The adoption of DTMCs implies that the modeled system meets, with some tolerable approximation, the Markov property–described later on in Section 2.1. This issue can be easily verified as discussed in [6, 14].

As for most design approaches based on DTMCs (consider for example [14]), in our work we assume that the model depscribes behaviors that depend on interaction profile and failure probabilities, which are used to label DTMC transitions. These values may be hard to predict at design time. In practice, a software designer may rely on estimates for interaction and failure probabilities, gathered by running instances of similar systems, as discussed in [12]. Some of these values, in addition, may change over time after the system has been developed and deployed.

We make the assumption that, through careful design-time analysis, we can restrict run-time variability to a subset of environment parameters. Precisely, we assume that (1) we can anticipate the variable transitions in the model and (2) they are a small fraction of the total number of transitions. These assumptions are valid in many practical cases. If they do not hold, our approach may still be applied, but simply would not yield its expected benefits in terms of speed-up of run-time verification.

In the next section we briefly introduce DTMCs. Afterwards, we describe PCTL [2], a probabilistic temporal logic, adopted here to express reliability requirements.

2.1 Discrete Time Markov Chains

DTMCs are defined as state-transition systems augmented with probabilities. *States* represent possible configurations of the system. *Transitions* among states occur at discrete time and have an associated probability. DTMCs are discrete stochastic processes with the Markov property, according to which the probability distribution of future states depend only upon the current state.

Formally, a (labeled) DTMC is tuple (S, S_0, P, L) 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 ($\sum_{s' \in S} P(s,s') = 1 \;\; \forall s \in S$). 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* which are true in that state.

In this paper we will implicitly extend this definition by also allowing transitions to be labeled with variables (in the range 0..1) instead of constants. A state $s \in S$ is said to be an *absorbing state* if $P(s,s) = 1$. If a DTMC contains at least one absorbing state, the DTMC itself is said to be an *absorbing DTMC*.

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

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

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 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 (representing 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 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.

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. Once the model is in place, we may be interested in estimating the probability of reaching an absorbing state or in stating the property that the probability of reaching an absorbing failure state should be less than a certain threshold. In the next section we discuss how these and other interesting properties of systems modeled by a DTMC can be expresses and how they can be evaluated.

Let us consider for example the DTMC in Figure 2, which represents a system sending authenticated messages over the network. States 5, 6, and 7 are absorbing states; states 6 and 7 represent failures associated respectively to the authentication and to message sending. We use variables as transition labels to indicate that the value of the corresponding probability is unknown, and may change over time.

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

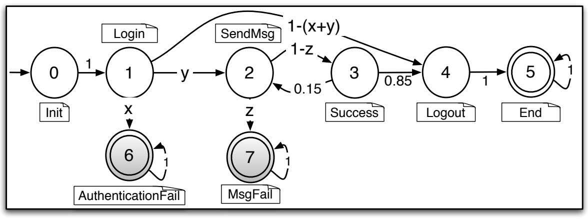


Figure 2: DTMC Example.

$$Q = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & y & 0 & 1-x-y \\ 0 & 0 & 0 & 1-z & 0 \\ 0 & 0 & 0.15 & 0 & 0.85 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 0 & 0 & 0 \\ 0 & x & 0 \\ 0 & 0 & z \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

This is a toy example that we use to introduce the proposed approach. However, the concepts described hereafter apply to real systems which might have thousands of states and failures, as we discuss in Section 4.

2.2 PCTL and Reliability Properties

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 PCTL [2], a logic which can be used to express a number of interesting reliability properties.

PCTL is a logic language inspired by CTL [2]. In place of the existential and universal quantification of CTL, PCTL provides the probabilistic operator $\mathcal{P}_{\bowtie p}(\cdot)$, where $p \in [0,1]$ is a probability bound and $\bowtie \in \{\leq, <, \geq, >\}$.

PCTL is defined by the following syntax:

$$\Phi ::= true \mid a \mid \Phi \,\wedge\, \Phi \mid \neg\, \Phi \mid \mathcal{P}_{\bowtie p}\ (\varphi)$$
$$\varphi ::= X\ \Phi \mid \Phi\ U\ \Phi \mid \Phi\ U^{\leq t}\ \Phi$$

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 over the set of all possible paths originating in the state where they are evaluated.

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 formal definition of how to compute $Pr(s \models \psi)$ is presented in [2]. The intuition is that its value corresponds to the fraction of paths originating in s and satisfying ψ over the entire set of 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:

$\pi \models X\Phi$ iff $\pi[1] \models \Phi$

$\pi \models \Phi U \Psi$ iff $\exists j \geq 0.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))$

$\pi \models \Phi U^{\leq t} \Psi$ iff $\exists 0 \leq j \leq t.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi))$

PCTL is an expressive language that allows many interesting reliability-related properties to be specified. A taxonomy of all possible reliability properties is out of the scope of this paper. The most important case is a *reachability property*. A reachability property states that a state where a certain characteristic property holds is eventually reached from a given initial state. In most cases, the state to be reached is an absorbing state. Such state may represent a *failure state*, in which a transaction executed by the system modeled by the DTMC eventually terminates, or a *success state*. Reachability properties are expressed as $\mathcal{P}_{\bowtie p}(true\ U\ \Phi)$[1], which expresses the fact that the probability of reaching any state satisfying Φ has to be in the interval defined by constraint $\bowtie p$. Φ is assumed to be a simple state formula that does not include any nested path formula. In most cases, it just corresponds to the atomic proposition that is true 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. For example, referring to the example in Figure 2, we express the following reliability requirements:

- **R1**:*"The probability that a MsgFail failure happens is lower than* 0.001*"*
- **R2**:*"The probability of successfully sending at least one message for a logged in user before logging out is greater than* 0.001*"*
- **R3**:*"The probability of successfully logging in and immediately logging out is greater than* 0.001*"*
- **R4**:*"The probability of sending at least 2 messages before logging out is greater than or equal to* 0.001*"*

These requirements can be translated into PCTL as shown in Table 1. Notice that **R1** is an example of reachability property. Also notice that these requirements have different sets of initial states: **R1**, **R3**, and **R4** must be evaluated starting from state 0 (i.e., $S_0 = \{0\}$) while **R2** must be evaluated starting from state 1.

In the next section we present a mathematical procedure to compute a (symbolic) formula (i.e., an analytic expression) for the properties we want to verify at run time. We start by analyzing reachability properties and then we progressively show how to cover all of PCTL formulae.

3. THE APPROACH

In this section we illustrate how PCTL formulae may be pre-computed at design time. Pre-computation produces a formula for each PCTL property. The formula is an analytic expression that contains variables which become known at run-time. Variables correspond to transition probabilities that are unknown (or uncertain) at design time. Note that there must be at least two variable transitions exiting a state, since the sum of probabilities must be 1. In general, we may assume that if there is a variability, all the transitions exiting a node are variable. We refer to such states as *variable states*. We start this section by discussing how reachability formulae may be pre-computed. We then discuss how to extend our approach to cover the entire PCTL.

[1]Note that this is often expressed as $\mathcal{P}_{\bowtie p}F\Phi$, using the *finally* operator.

Table 1: Requirements translation in PCTL.

Req.	PCTL
R1	$P_{\leq 0.001}(true\ U\ s=7) = P_{\leq 0.001}(F\ s=7)$
R2	$P_{\geq 0.001}(1 \leq s \leq 2\ U\ s=3)$
R3	$P_{\geq 0.001}(X\ s=4)$
R4	$P_{\geq 0.001}(1 \leq s \leq 3\ U^{\leq 5}\ s=4)$

3.1 Reachability Formulae

The most commonly studied property for reliability analysis is the probability of reaching a certain state, which typically represents the success of the system or some failure condition. Both success and failure are modeled by absorbing states. The reachability formula in this case has the following form: $\mathcal{P}_{\bowtie p}Fl$, where l is the label of the target absorbing state. Let us start our discussion by showing how to pre-compute at design time a reachability formula for an absorbing state of a DTMC.

For an absorbing DTMC, the matrix $I - Q$ has an inverse N and $N = I + Q + Q^2 + ... = \sum_{i=0}^{\infty} Q^i$ [15]. The entry n_{ij} of N represents the expected number of times the Markov chain reaches state s_j, given that it started from state s_i, before getting absorbed. Instead, q_{ij} represents the probability of moving from the transient state s_i to the transient state s_j in exactly one step.

Given that $Q^n \to 0$ when $n \to \infty$ (as discussed in Section 2.1), the process will always be absorbed with probability 1 after a large enough number of steps, no matter which state it started in. Hence, our interest is to compute the probability distribution over the set of absorbing states. This distribution can be computed in matrix form as:

$$B = N \times R$$

where r_{ik} is the probability of being absorbed in state s_k given that the process started in state s_i.

B is a $t \times r$ matrix and it can be used to evaluate the probability of each termination condition starting from any DTMC state as an initial state. In particular the element b_{ij} of the matrix B represents the probability of being absorbed into state s_j given that the execution started in state s_i.

The design-time computation of an entry b_{ij} in general can only be done symbolically, since variable states may be traversed to reach state s_j. Let us evaluate the complexity of such computation. Inverting matrix $I - Q$ by means of the Gauss-Jordan elimination algorithm [1] requires t^3 operations. The computation of the entry b_{ij} once N has been computed requires t more products, thus the total complexity is $t^3 + t$ algebraic operations on polynomials. The computation could be further optimized by exploiting the sparsity of $I - Q$. Notice that the symbolic nature of the computation makes the design-time phase quite costly [16].

The complexity can be significantly reduced if the number of variable components c is small and the matrix describing the DTMC is sparse, as very frequently happens in practice. Let $W = I - Q$. The elements of its inverse N are defined as follows:

$$n_{ij} = \frac{1}{det(W)} \cdot \alpha_{ji}(W)$$

where $\alpha_{ji}(W)$ is the cofactor of the element w_{ji}. Thus:

$$b_{ik} = \sum_{x \in 0..t-1} n_{ix} \cdot r_{xj} = \frac{1}{det(W)} \sum_{x \in 0..t-1} \alpha_{xi}(W) \cdot r_{xj}$$

Computing b_{ik} requires the computation of t determinants of square matrices with size $t-1$. Let τ be the average number of outgoing transitions from each state ($\tau << n$ by assumption). Each of the determinants can be computed by means of Laplace expansion. Precisely, by expanding first the c rows representing the variable states (each has τ symbolic terms), we need to compute at most τ^c determinants and then linearly combine them. Each submatrix of size $t-c$ does not contain any variable symbol, by construction, thus its determinant can be computed with $(t-c)^3$ operations among constant numbers (LU-decomposition), thus much faster than the corresponding ones among polynomials. The final complexity is thus:

$$\tau^c \cdot (t-c)^3 \sim \tau^c \cdot t^3 \qquad (1)$$

which significantly reduces the original complexity and makes the design-time pre-computation of reachability properties feasible in a reasonable time, even for large values of t.

As a point of comparison, the computation of reachability properties performed by probabilistic model-checkers is based on the solution of a system of n equations in n variables [2], which has, in a sequential computational model, a complexity equal to n^3 [4].

Summing up, we discussed the computation of properties in the form $\mathcal{P}_{\bowtie p}(Fs_k)$, where s_k is an absorbing state, starting in any initial transient state of the system[2]. With this procedure, it is possible to obtain closed formulae for a number of interesting reliability properties.

For example, evaluating **R1** on our example system, that is the probability of reaching the state *MsgFail* failure in any number of execution steps corresponds to evaluating b_{07} as:

$$\textbf{R1:} \qquad \frac{(yz)}{(0.85 + 0.15z)} \leq 0.001$$

Let us now consider the computation of the probability of successfully reaching a certain state s_j that is not an absorbing state[3]. The quantity we are interested in is f_{ij}, the probability of ever making a transition into state s_j given that the process started in state s_i (s_i can be any transient state taken as initial state). Formally, let f_{ij}^n represent the probability of moving from the transient state s_i to the transient state s_j for the first time in exactly n steps:

$$\begin{cases} f_{ij}^0 &= 0 \; \forall i \neq j; f_{ij}^0 \;= 1 \; \forall i = j \\ f_{ij}^n &= \; Pr\{X_n = s_j \; \wedge X_k \neq s_j \; \forall \; 1 \leq k \leq n-1 | X_0 = s_i\} \end{cases}$$

where X_k is the state of the system at execution step k. Then:

$$f_{ij} = \sum_{n=0}^{\infty} f_{ij}^n$$

It is possible to compute the value f_{ij} from the matrix N by conditioning the entry n_{ij}–the expected number of times the DTMC on whether state s_j is ever entered [27]:

$$n_{ij} = n_{jj} f_{ij}$$

thus:

$$f_{ij} = \frac{n_{ij}}{njj} = \frac{\alpha_{ji}(W)}{\alpha_{jj}(W)}$$

Hence, computing the probability of moving from a transient state s_i to a transient state s_j is reduced to the computation of the determinants of two matrices with size $t-1$. Again, by the fact that only a few rows of N are symbolic (i.e. only a few states are variable), the actual complexity of the computation is approximately the same as in expression 1.

The approach we described so far supports the definition of properties which represent requirements that speak about *"the probability of reaching state s_j without reaching any failure"* or *"the probability of a successfully performing a certain operation or service"*. In our example, the probability of reaching the *Logout* state 7 after any number of steps[4] corresponds to the entry $f_{04} = \frac{0.85-0.85x+0.15z-0.15xz-yz}{0.85+0.15z}$.

Once again, we stress that our approach is especially practical under the assumption that the number of parameters of the system is reasonably small. Design-time computational effort could be further reduced by adopting state-of-the-art (parallel) matrix calculus algorithms [20].

3.2 Extending to Full PCTL

In the previous section we described a solution limited to reachability properties. Even though reachability represents the most widely adopted pattern for reliability analysis, it does not cover all the requirements that engineers need to express for real-world applications; consider for example requirements **R2**–**R4** of our example. In this section, we incrementally extend the approach to handle all of PCTL.

Notice that reachability properties correspond to restricted until formulae $\mathcal{P}_{\bowtie p}(\Phi_1 \cup \Phi_2)$ where: (1) Φ_1 corresponds to *true* (i.e., Φ_1 is satisfied in any state) and (2) Φ_2 does not include any nested path formula (we refer to this subset of PCTL which does not allow the nesting of path operators as the *flat subset*). Sections 3.2.1 and 3.2.2 discuss our approach for each PCTL operator in the case of flat formulae. Section 3.2.3 then describes how to extend the approach to nested formulae, thus covering the entire PCTL.

3.2.1 Flat Until formulae

The procedure to support requirements in the (flat) form $\Phi_1 \cup \Phi_2$ relies on: (1) reducing them to reachability properties and (2) applying the technique described in sect. 3.1.

[2]Actually we discussed the computation of the probability associated with the property, to which the constraint $\bowtie p$ has to be applied.

[3]Our mathematical description follows the treatment in [27], to which we direct the reader for a more detailed discussion.

[4]Note that this probability is not what is needed for **R3**, which in turn requires to reach logout in a single step. See Section 3.2.2.

The reduction procedure starts with the construction of a DTMC $\bar{D}$ defined as follows. We refer to the set of states of $\bar{D}$ as $S_{\bar{D}}$. The set is the union of three non-overlapping subsets, S_{goal}, $S_{\neg goal}$ and $S_{transient}$, respectively defined as: (1) all the absorbing states of the original DTMC in which Φ_2 is true plus an additional auxiliary state s_{goal}, (2) all the absorbing states of the original DTMC in which Φ_2 is false plus an additional auxiliary state $s_{\neg goal}$, and (3) all the remaining states of the original DTMC: $S_{transient} = S/\{S_{goal} \cup S_{\neg goal}\}$. The following algorithm defines the transitions of $\bar{D}$ starting from the transitions of the original DTMC:

1. delete all the outgoing transitions from all the transient states of the original DTMC in which $\neg(\Phi_1 \vee \Phi_2)$ is *true* and add a single outgoing transition directed to $s_{\neg goal}$ with a labelling probability equal to 1.

2. attach to all the transient states of the original DTMC where Φ_2 is true a single outgoing transition directed to s_{goal} with a labelling probability equal to 1.

Recalling our example of Figure 2 and considering requirement **R2** we have that:

$$S_{goal} = \{s_{goal}\}$$
$$S_{\neg goal} = \{s_{\neg goal}, 5, 6, 7\}$$
$$S_{transient} = \{0, 1, 2, 3, 4\}$$

Figure 3 illustrates the resulting DTMC $\bar{D}$.

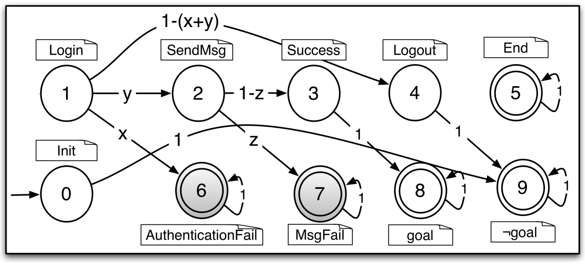


Figure 3: Resulting DTMC $\bar{D}$.

Goal states in S_{goal} represent the satisfaction of the formula. In fact, all the path formulae in the form $\Phi_1\ U\ \Phi_2$ are satisfied in all the states in which Φ_2 is true and those states, in $\bar{D}$, are directly connected to goal states. Moreover, the predecessor states on any path that leads to a state where Φ_2 is true are such that Φ_2 is false (otherwise from there we would reach S_{goal} with probability 1) and Φ_1 is true (otherwise it would have been deleted by step 1 of our algorithm). Conversely, states in $S_{\neg goal}$ can be reached directly, and with probability 1, by all the states in which the formula is not satisfied, i.e. $\neg(\Phi_1 \vee \Phi_2)$ is *true*. Hence, we reduced the evaluation of a general Until property to a reachability property such as: $\mathcal{P}_{\bowtie p}(F\ s \in S_{goal})$.

For example, evaluating **R2** on the original DTMC is equivalent to evaluating the reachability property $\mathcal{P}_{\bowtie p}$(F $s \in S_{goal}$) over $\bar{D}$, which corresponds to the entry b_{18}, where 8 is the index of state s_{goal}, computed over the transition matrix of $\bar{D}$:

$$\textbf{R2:} \qquad y - yz \geq 0.001$$

3.2.2 Flat Next and Bounded Until formulae

Let us consider again the flat subset of PCTL and let us focus on the pre-computation of formulae that use the Next and the Bounded Until operators, which require analyzing paths of finite length.

The set of paths to be considered in order to evaluate the formula $X\Phi_1$ in state s_i is composed by all 1-step long paths exiting s_i. The maximum size of such set is n (i.e., the number of states), which is also the worst-case complexity of our design-time pre-computation procedure. The probability that $s_i \models X\Phi_1$ is:

$$Pr(X\Phi_1) = \sum_{s_j \models (\Phi_1)} p_{ij}$$

For example, in order to evaluate **R3**, we first notice that $s = 4$ is satisfied only in state *Logout*. Thus, the probability of satisfying $s = 4$ in exactly one step from state 1 (*Login*) is expressed by the formula $1 - x - y$, and **R3** becomes

$$\textbf{R3:} \qquad 1 - x - y \geq 0.001$$

A similar procedure applies to the Bounded Until. A path originating in s_i, which satisfies the formula $\Phi_1\ U^{\leq v}\ \Phi_2$, at a certain step $k \leq v$ satisfies Φ_2 and for all the states $l < k$ satisfies Φ_1. We therefore need to consider all the paths with length $k \leq v$.

If we exploit the DTMC $\bar{D}$ built as explained in the previous section, all these paths correspond to paths of $\bar{D}$ whose length is equal to exactly $v + 1$ and which reach a transient state satisfying Φ_2 in v steps and then end in any of the states in S_{goal}. Indeed, if a path reaches an absorbing state after k steps, it remains in that state with probability equal to 1, thus the tail of the path will be composed of $v + 1 - k$ self-transitions with probability 1 exiting a state in S_{goal}.

The probability of moving in $v + 1$ steps from a state s_i to a state s_j corresponds to the entry p_{ij}^{v+1} of the $(v+1)$-th power of the transition matrix P of the DTMC $\bar{D}$. Hence:

$$Pr(\Phi_1\ U^{\leq v}\ \Phi_2) = \sum_{s_j \in S_{goal}} p_{ij}^{v+1}$$

In our example, let us consider **R4**. After constructing $\bar{D}$, it is possible to compute the probability of reaching any of the goal states from the *Login* state. In this case there is only one goal state s_{goal} because the formula $s = 4$ is false in any other absorbing state. Thus, we need to compute the entry p_{18}^{6}, where 8 is the index of s_{goal}, and such a value is:

$$\textbf{R4:} \qquad 1 - x - y + 0.85y(1-z) + 0.1275y(1-z)^2 \geq 0.001$$

To assess design-time complexity in this case, we should consider that we need to compute the matrix P^{v+1}. Since the complexity of a matrix dot product is approximately n^3 [8], a non-optimized algorithm has complexity vn^3.

At runtime, both Next and Bounded Until only require to evaluate a polynomial, by substituting values to transition variables, as in the case of reachability formulae.

3.2.3 Handling Nested Path Formulae

We have shown that for all kinds of flat PCTL formulae it is possible to generate a corresponding symbolic expression at design time. In the case of nested path operators, that is for formulae $\mathcal{P}_{\bowtie p}(\psi)$ where at least one subformula of ψ is again a path formula, some information needed to compute the expression might be unavailable at design time.

For example, the actual violation of requirements **R1-R4** will be known only at run time, when parameter values will be available. If any of those formulae is, for example, part of the right-hand operand of an Until formula, the construction of the reduced DTMC $\bar{D}$ must be delayed to run time, when the set of states which satisfies Φ_2 (and Φ_1) becomes known from the values bound to the parameters. Thus, in general, to evaluate a formula with nested $\mathcal{P}_{\bowtie p}(\cdot)$ operators, we need to know in which states its subformulae are satisfied, and the definition of this set may depend on parameter values. The same observation can be applied recursively to subformulae of a subformula, until we reach a flat formula, for which we can immediately construct an equivalent expression.

To develop a solution, we need a way to delay the evaluation of a formula to run time, when all of its subformulae will be already evaluated providing the missing knowledge. In order to support this process, we add some extra parameters at design time, which will account for the lack of information concerning subformulae's satisfaction. Those parameters constitute a scaffolding introduced at design time, which is removed as subformulae are evaluated.

Let us first focus on Until formulae, like $\Phi_1 U \Phi_2$. Recalling the procedure in Section 3.2.1, the construction of $\bar{D}$ requires two basic operations (besides the introduction of absorbing states s_{goal} and $s_{\neg goal}$): (1) replacing all outgoing transitions from states where $\neg(\Phi_1 \vee \Phi_2)$ holds by a single one toward $s_{\neg goal}$, and (2) replacing all outgoing transitions from states where Φ_2 is true by a single one toward s_{goal}. From a mathematical viewpoint, deleting a transition is equivalent to labeling it with 0 probability. By multiplying each non-zero transition p_{ij} of the original DTMC by a coefficient $m_{ij} \in \{0,1\}$, it is thus possible to delay the decision whether a transition should be deleted or not by later assigning 0 or 1 to the corresponding coefficient m_{ij}. To construct $\bar{D}$, we also need to be able to connect a transient state to s_{goal} or $s_{\neg goal}$. In order to do that, we complete our scaffolding by introducing two transitions, labelled $a_{is_{goal}}$ and $a_{is_{\neg goal}}$, to connect each state i to the newly introduced s_{goal} and $s_{\neg goal}$, respectively. These labels can be assigned at run time value 1 in case the construction of $\bar{D}$ requires to connect the transient state i to s_{goal} or $s_{\neg goal}$, respectively.

By applying the scaffolding procedure to our example, we obtain for matrices Q and R:

$$Q = \begin{pmatrix} 0 & m_{0,1} & 0 & 0 & 0 \\ 0 & 0 & m_{1,2}y & 0 & m_{1,4}(1-x-y) \\ 0 & 0 & 0 & m_{2,3}(1-z) & 0 \\ 0 & 0 & m_{3,2}0.15 & 0 & m_{3,4}0.85 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 0 & 0 & 0 & a_{0,8} & a_{0,9} \\ 0 & m_{1,6} \times x & 0 & a_{1,8} & a_{1,9} \\ 0 & 0 & m_{2,7} \times z & a_{2,8} & a_{2,9} \\ 0 & 0 & 0 & a_{3,8} & a_{3,9} \\ m_{4,5} & 0 & 0 & a_{4,8} & a_{4,9} \end{pmatrix}$$

The last phase of the decision procedure in Section 3.2.1 requires to sum the probabilities of reaching every absorbing state in S_{goal}. This set will be known at run time, when the subformula Φ_2 can be evaluated. At design time, we simply compute the entire matrix B (the probabilities of going from each transient state to each absorbing state) from the original DTMC, instrumented with the just presented scaffolding. At run time, when the set S_{goal} becomes known, we assign a value to m_{ij} and a_{ij} coherently with what we did in Section 3.2.1 and sum all the b_{ij} from the transient i in which the formula is being evaluated to a state $j \in S_{goal}$.

Concerning Next and Bounded Until operators, the methods described in Section 3.2.2 are still valid, though they have to be applied to the matrix P instrumented with the scaffolding, so that, at run time, it is possible to assign values to m_{ij} and a_{ij} to account for the newly acquired knowledge.

In the case of nested path operators, at design time, it is not possible to exploit the mixed–symbolic (expanded via Laplace) and numeric–computation of b_{ij} presented in Section 3.1. Nevertheless, being m_{ij} and a_{ij} boolean, instead of floating point multiplications with constant values it is possible to use faster bitwise AND, while multiplying a polynomial by 1 has no cost and multiplying it by 0 trivially returns 0. Thus the introduction of the scaffolding does not affect significantly the complexity of the design-time analysis we illustrated in Section 3.1.

At run time, in the case of nested path operators, the PCTL formula cannot be evaluated in a single step, as we did for flat formulae. Nevertheless the number of expressions to compute is *linear* in the number of path operators, and each formula has to be evaluated at most for all transient states. Each evaluation still works on a polynomial. The run-time efficiency gain over conventional model checkers is discussed next for all PCTL formulae.

4. VALIDATION

The main goal of this work is to find an efficient way of computing reliability properties in frequently changing environments. We described a solution that performs a (design-time) computationally expensive derivation of verification formulae that can be evaluated very efficiently at run time, when parameter values become known. The approach fits the very frequent situation in which time consumption during development is not critical, but run-time evaluation of verification formulae is subject to tight time constraints.

Concerning run-time evaluation[5], we compared the performance of a simple Java/C prototype implementation of our approach with the outputs produced by two widely used probabilistic model checkers (PRISM [18] and MRMC [21]) and with a numerical computation of results of formulae by Matlab. All the tools were required to produce an approximation of at most 10^{-15} and to run with its default solution strategy.

The test suite is composed of 127 samples. Each sample is a randomly generated DTMC with a number of states varying from 50 to 500 (with step 50), with 2 absorbing states (namely *correct completion* and *failure*). Each state has a number of outgoing transitions randomly sampled from a Gaussian distribution with mean 10 and standard deviation 2. The number of variable states is 4 for all the samples, and when a state is variable so are all its outgoing transitions. We did not consider the process spawning time and we report as execution time the one provided by each tool. The execution environment is a dedicated machine with 2 Intel(R) Xeon(R) CPU E5530 2.40GHz and 8Gb of RAM. The operating system is Ubuntu Server 2.6.24-24, 64bit. Matlab version is 2008a (release 7.6.0.324), PRISM version 3.3.1 and MRMC 1.4.1 both compiled at 64bit with default compiling options. Our prototype generates the input files for all of these tools and a C program computing our formulae. Concerning PRISM, here we consider model-checking time only, which does not take into account the model construction

[5]The software used for the experimental assessment and the datasets are available at http://home.dei.polimi.it/filieri/icse2011

time [18] (up to 15 secs for a 500 states DTMC).

The empirical validation focused on reachability formulae. In practice, most useful reliability properties are expressed as reachability formulae. In addition, as we showed in Section 3, reachability is at the core of analysis for also other PCTL properties.

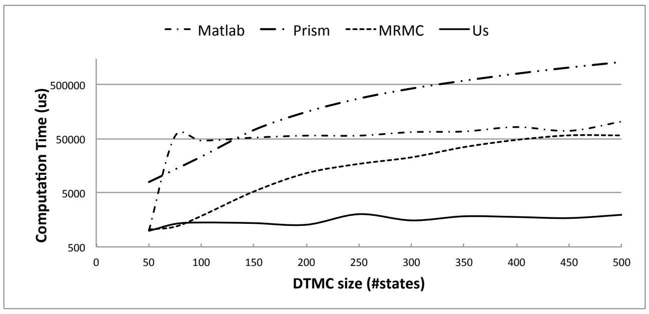


Figure 4: Runtime Verification.

The result of our comparison is shown in Figure 4 (we use a logarithmic scale for time). Computation time growth quickly as the size of the DTMC raises, both for PRISM and MRMC. PRISM exhibits serious performance problems, probably due to the fact that it is unable to take advantage of its symbolic engine [24], which instead performs pretty well in the reduction of complex PCTL formulae to reachability problems. The state-space reduction approach adopted by MRMC [22] seems to be more successful for our samples. PRISM becomes an order of magnitude worse than our approach after 75 states, while MRMC after 200 states. With 500 states PRISM takes an order of 10^6 μs and MRMC still 10^4 μs. We also used Matlab to compute the same procedure based on linear algebra that was adopted in our methodology, but with numerical methods. Input matrices were declared as sparse, so that the Matlab numerical engine chooses the best algorithm to perform computation.

The runtime performance of our tool is close to a constant for reachability formulae. Independent of the size of the input DTMC, computation time is in the order of 10^3 μs with a maximum of 2014 μs, an average of 1565.27 μs and a standard deviation of 322.77 μs. Fluctuations in the values are due to the topology of the matrices, which can lead to longer or shorter polynomial forms in our mathematical formulae, depending on how they scatter variable symbols during computation. The gap between 450 and 500 is essentially due to some outliers in the dataset, whose effect can be reduced by extending the sample set (available on the web). In any case, the number of possible combination of those symbols is always bounded. Remember that we are considering the case in which only a few states are variable, and their number is the most influent parameter for our complexity, determining how many variables appear in our mathematical formulae. Our approach is independent of the size of the DTMC, and the resulting mathematical formula can be implemented directly in any programming language, without any need to integrate with external tools or libraries. Notice that the mathematical formula we produce is not optimized neither at the computation level, by grouping terms of factorizing polynomials, nor at the compilation level (e.g., via optimization flags for mathematical computation as in gcc[6]).

The price for such a fast run-time evaluation has to be paid at design time, though only once. As explained in Section 3, the three parameters on which design-time computation depends are: (1) the size of the system, (2) the number of variable states, and (3) the number of transitions outgoing from variable states. Our prototype design-time symbolic manipulation engine is not optimized; thus the execution times reported hereafter are just an indicative order of magnitude of the complexity scale of the problem in a non-parallel execution environment. All the following experiments were conducted in the same execution environment described above for run-time performance evaluation. Execution on a parallel machine might bring a drastic reduction of execution time.

All DTMCs in the following test suites have 2 absorbing states representing successful termination and failure of a hypothetical system. The DTMC are generated randomly with 2 variable states and an average of 10 outgoing transitions (standard deviation 2). As before, if a state is variable, so are all of its outgoing transitions.The reliability property is expressed as the reachability of a successful termination state.

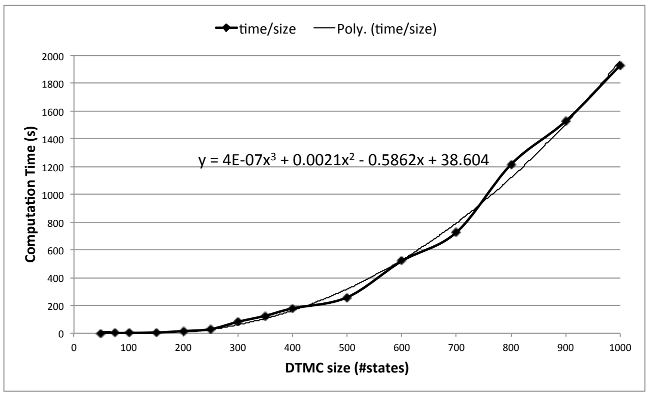


Figure 5: Precomputation time over DTMC's size.

Figure 5 describes how pre-computation time varies according to the size of the system. The grey line shows an interpolated trend-line whose equation is also shown in the figure. The expected complexity in this case was n^3, where n is the number of states in the DTMC. As a matter of fact, the use of Jama[7] for the numerical part of the computation slightly reduced the expected computation time, which is still a low-order polynomial.

The number of variable states is the most critical parameter for design-time pre-computation. Figure 6 reports precomputation time for a number of variable states from 1 to 5.

All the sample DTMCs (3 per class) are composed by 200 states, each of those with an average of 10 outgoing transitions (standard deviation 2). The time axis in Figure 6 is in logarithmic scale. The diagram shows that time complexity is exponential and roughly in the order of 14^c, where c is the number of variable states, which is consistent with what one expects considering the randomness of the number of

[6]http://gcc.gnu.org

[7]http://math.nist.gov/javanumerics/jama

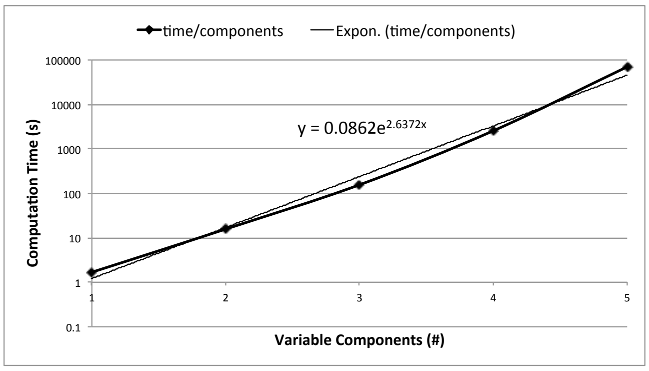


Figure 6: Precomputation time over variable states.

transitions of the input samples.

The last parameter which affects design-time pre-computation is the average number of transitions from variable states. Figure 7 shows the results of our experiments. For simplicity we sampled the number of outgoing transitions for all the states (both variable and not) from a Gaussian distribution having as mean the number of transitions on the abscissa and standard deviation equal to mean/4 (rounded to the closest integer). The DTMC has 200 states, 2 of which are variable. Hence the expected complexity is in the order of τ^2, which was confirmed by the empirical results.

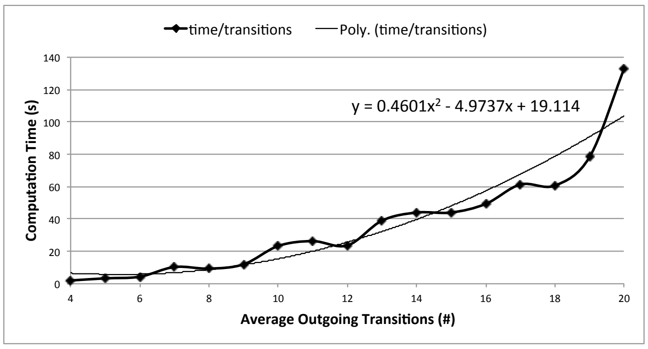


Figure 7: Precomputation time over τ.

5. RELATED WORK

In this paper we focused on reliability requirements for DTMC-based models. Based on the seminal work described in [6], DTMCs become the most widely adopted modeling formalism to deal with reliability at architecture level. A number of approaches have been proposed in this direction [19, 5]. The latter presents a framework for component reliability prediction whose objective is to construct and solve a stochastic reliability model allowing software architects to explore competing alternatives. Specifically, the authors tackle the definition of reliability models at architectural level and the problems related with parameter estimation.

Besides the basic estimation of failure occurrence, there are a number of advanced reliability properties that can be analyzed by means of DTMC-based techniques, e.g. failure propagation [9] and failure evolution and transformation in presence of multiple failure modes [13].

An important problem, shared by all design approaches based on DTMCs, is how to get interaction and failure probability values [14]. Many solutions came out from the research community, from accelerated test [17] to mining of bug repositiories [26], up to the estimation of failure probabilities even in case no failure has been observed [25], and so and so on. To make our methodology worthy, one must be able to estimate relevant system parameters on-the-fly, by monitoring the system. Many methods exist to support reasoning on non-functional properties of software based on models that are analyzed at run-time by relying on monitoring such as [3, 12, 28, 29], but for all of them the tighter bottleneck is how to realize that a requirement is being violated in a time short enough to allow effective reactions.

Probabilistic model-checking plays a crucial role in evaluating reliability properties, typically expressed in PCTL, over DTMC models of the running system. In practice, however, they require minutes or more to evaluate properties over large models, thus hindering planning and reconfiguration capabilities that must respond to tighter time bounds. In [11] Daws proposed a procedure to first convert the DTMC into a finite automaton from which it is possible to obtain a corresponding regular expression. This expression can be evaluated to a mathematical formula which represents any arbitrary reachability property. Daws' approach is restricted to formulae without nested probabilistic operators and the outcoming regular expression grows quickly with the number of states composing the DTMC ($n^{log(n)}$). In [16] Hahn et al. propose a refinement of the approach presented in [11] for reachability formulae which combines state space reduction techniques and early evaluation of the regular expression in order to improve actual execution times when only a few variable parameters appear in the model. The improvement in [16] requires n^3 arithmetic operations among polynomials, performing better than [11] in most practical cases, although still leading to a $n^{log(n)}$ long expression in the worst case. As opposed to our approach, [16] only deals with reachability properties. For reachability properties, by applying our approach in a sequential environment and considering each parametric transition as a polynomial expression, one can obtain approximately the same complexity as [16] without resorting to particularly efficient determinant computation methods [20]. For example, by applying the Coppersmith–Winograd algorithm we could reduce complexity to $n^{2.376}$. Parallelization, as well as exploitation of sparsity of matrix W (see Section 3.1) would lead to an even higher improvement in design-time computation performance.

6. CONCLUSIONS AND FUTURE WORK

We addressed the problem of efficient run-time model checking of reliability models expressed as DTMCs. We provided a mathematical approach that divides the model checking process in two steps to be computed respectively at design time and run time, improving considerably the run-time performance of analysis. The approach, which is particularly valuable for systems with a limited number of variability points, provides full support to PCTL and is particularly efficient for reachability properties.

We implemented the approach in a prototype tool, which will be made available as an open source artifact. We performed extended simulations, but for space reasons we could only report on selected cases. We focused on comparing our solution with state-of-the-art tools, like PRISM [18] and MRMC [21]. An empirical comparison with PARAM[8], the tool implementing the approach of [16], concerning reachability properties is also planned after we will develop an optimized implementation of our tool. Our approach is a solution based on linear algebra and well known algorithms, that can be highly parallelized to make our approach more efficient. In the future, we plan to complement simulation-based validation with real-world case studies to stress the scalability and effectiveness of the approach. In addition, we plan to reduce design-time complexity by means of state space reduction and partial order reduction techniques.

Finally, the approach might be extended both to other logics, such as PCTL*, and to other Markov models, such as Continuous Time Markov Chains for performance analysis.

Acknowledgments

The authors thank Ilenia Epifani for her continuous support on probability theory and Holger Hermanns for useful comments on an earlier draft. This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

7. REFERENCES

[1] S. C. Althoen and R. McLaughlin. Gauss-jordan reduction: A brief history. *The American Mathematical Monthly*, 94(2):130–142, 1987.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[3] L. Baresi and S. Guinea. Towards dynamic monitoring of ws-bpel processes. *ICSOC*, 2005.

[4] A. Bojanczyk. Complexity of solving linear systems in different models of computation. *SIAM Journal on Numerical Analysis*, 21(3):591–603, 1984.

[5] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik. Early prediction of software component reliability. In *ICSE, Leipzig, Germany, May 10-18, 2008*, pages 111–120. ACM, 2008.

[6] R. C. Cheung. A user-oriented software reliability model. *IEEE Trans. Softw. Eng.*, 6(2):118–125, 1980.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* Springer, 1999.

[8] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM J. Comput.*, 11(3):472–492, 1982.

[9] V. Cortellessa and V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In H. W. Schmidt, I. Crnkovic, G. T. Heineman, and J. A. Stafford, editors, *CBSE*, volume 4608 of *LNCS*, pages 140–156. Springer, 2007.

[10] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4), 1995.

[11] C. Daws. Symbolic and parametric model checking of discrete-time markov chains. *ICTAC*, pages 280–294, 2005.

[12] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE*, 2009.

[13] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola. Reliability analysis of component-based systems with multiple failure modes. In *CBSE*, pages 1–20, 2010.

[14] K. Goseva-Popstojanova and K. S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179 – 204, 2001.

[15] C. Grinstead and J. Snell. *Introduction to Probability.* Amer Mathematical Society, 1997.

[16] E. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric markov models. *Model Checking Software*, pages 88–106, 2009.

[17] P. Heidelberger. Fast simulation of rare events in queueing and reliability models. *TOMACS*, 5(1):43–85, 1995.

[18] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. *TACAS*, 3920:441–444, 2006.

[19] A. Immonen and E. Niemela. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.

[20] E. Kaltofen and G. Villard. On the complexity of computing determinants. *Computational Complexity*, 13(3):91–130, 2005.

[21] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *QEST*, pages 243–244, Los Alamos, CA, USA, 2005. IEEE Computer Society.

[22] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance Evaluation*, In Press, Corrected Proof:–, 2010.

[23] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. *Formal Methods for Performance Evaluation*, pages 220–270.

[24] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 2324 of *Lecture Notes in Computer Science*, pages 113–140. Springer Berlin / Heidelberg, 2002.

[25] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Trans. Softw. Eng.*, 18(1):33–43, 1992.

[26] C. Rahmani, H. Siy, and A. Azadmanesh. An experimental analysis of open source software reliability. In *F2DA, Niagara Falls*, 2009.

[27] S. Ross. *Stochastic Processes.* Wiley New York, 1996.

[28] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE*, pages 371–380. ACM, 2006.

[29] T. Zheng, M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Trans. Softw. Eng.*, 34(3):391–406, 2008.

[8] http://depend.cs.uni-sb.de/tools/param