Supporting Self-adaptation via Quantitative Verification and Sensitivity Analysis at Run Time

Antonio Filieri, Giordano Tamburrelli, Carlo Ghezzi, *Fellow, IEEE*

Abstract—Modern software-intensive systems often interact with an environment whose behavior changes over time, often unpredictably. The occurrence of changes may jeopardize their ability to meet the desired requirements. It is therefore desirable to design software in a way that it can self-adapt to the occurrence of changes with limited, or even without, human intervention.

Self-adaptation can be achieved by bringing software models and model checking to run time, to support perpetual automatic reasoning about changes. Once a change is detected, the system itself can predict if requirements violations may occur and enable appropriate counter-actions. However, existing mainstream model checking techniques and tools were not conceived for run-time usage; hence they hardly meet the constraints imposed by on-the-fly analysis in terms of execution time and memory usage.

This paper addresses this issue and focuses on perpetual satisfaction of non-functional requirements, such as reliability or energy consumption. Its main contribution is the description of a mathematical framework for run-time efficient probabilistic model checking. Our approach statically generates a set of verification conditions that can be efficiently evaluated at run time as soon as changes occur. The proposed approach also supports sensitivity analysis, which enables reasoning about the effects of changes and can drive effective adaptation strategies.

Index Terms—Self-adaptive Systems, Software Evolution, Non-functional Requirements, Discrete-Time Markov models, Rewards, Software Reliability, Costs, Probabilistic Model Checking, Models at Runtime.

1 Introduction

Software is the driving engine of modern society. Most human activities—including 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. This is further exacerbated by the fact that modern software-intensive systems are often situated in complex contexts that can be hard or even impossible to fully understand and precisely describe at design time. Moreover, the context's behavior may change over time unpredictably, thus jeopardizing satisfaction of the desired requirements. Finally, these systems are continuously running and cannot be shut down to perform off-line reconfigurations or upgrades. Designing systems that can detect the occurrence of changes, reason about their effects, and possibly react to them in a self-adaptive manner has become a real challenge for software engineers. Systems of this kind are often called *self-adaptive* or *autonomic*.

Examples of potentially disruptive changes are:

1) Changes of location in the physical environment due to device mobility, which may affect connectivity conditions. This problem may occur in pervasive computing scenarios.
2) Changes in third-party services integrated in the system under consideration, which may cause unexpected misbehaviors of the integrated system. This problem may occur in the case of service-oriented systems.
3) Changes of clients' operational profiles, which may bring the running application into unexpected load conditions and cause unacceptable response times. This problem may occur in the case of user-intensive software applications.

• *A. Filieri is with the Reliable Software Systems group of the Institute for Software Technology at the University of Stuttgart, Stuttgart, Germany. E-mail: antonio.filieri@informatik.uni-stuttgart.de, G. Tamburrelli is with the Vrije University of Amsterdam, Netherlands. E-mail: g.tamburrelli@vu.nl, C. Ghezzi is affiliated to the Dipartimento di Elettronica, Informazione e Bioingegneria at Politecnico di Milano, Milan, Italy, E-mail: carlo.ghezzi@polimi.it*

4) Changes in the deployment environment, which may lead to violations of quality of service, such as response time. This problem may occur in cloud/service computing infrastructures.

Continuous changes in the environment typically affect cyber-physical systems, whose key components behave as sensors and actuators, used to sense and act upon the physical world. Last, but not least, the goals to be met and the requirements to be satisfied may also change over time.

In this work we ignore requirements evolution and focus instead on how to react to changes that may occur in the environment in which the application is embedded. In addition, we focus on changes that may affect the satisfaction of *non-functional requirements*, such as reliability, performance, and different kinds of cost-related requirements, such as energy consumption. We describe an approach that can predict possible failures caused by environment changes and thus self-adapt by triggering appropriate countermeasures. In the sequel we use the term *failure* to indicate a (non-functional) requirement's violation. For a further discussion about the definition of non-functional requirements and failures the reader can refer to [1].

Engineering self-adaptive systems calls for specific new approaches to the development and operation of software that guarantee lifelong requirements fulfillment in the presence of environmental changes. A self-adaptive systems must be able to (1) detect the relevant changes in the external world in which it operates, (2) reason about its own ability to continue to fulfill the requirements as a consequence of the detected changes, and (3) re-configure itself to guarantee a seamless adaptation to the new external conditions.

Software engineering for self-adaptive systems has been a growing research topic during the past decade and many promising results have been achieved. For a broad view of the area, the reader may refer to the series of SEAMS[1] workshops and symposia and the two Dagstuhl reports [2], [3]. Several promising approaches to software self-adaptation rely on the use of *models at run-time* [4]. Our work fully embraces this view. We keep models at run time and update them automatically as changes are dynamically discovered through monitoring (see [5], [6], [7], [8], [9]). Updated models can be used to reason about and predict possible requirements violations, which may then trigger proper adaptation strategies to steer system reconfigurations and prevent requirements violations. Conceptually, this framework establishes a *feedback control loop* between models and the running system. At run time, the system feeds data back to generate model updates. Adaptation thus becomes model-driven. This approach reflects the *autonomic control loop* advocated in [10] and illustrated (in slightly different terms) in Figure 1.

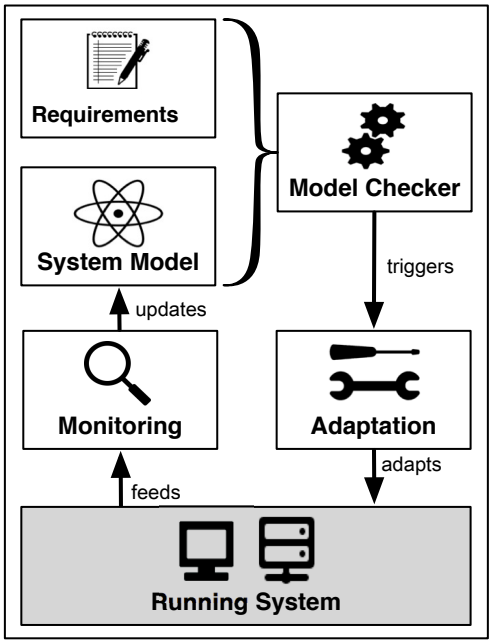


Fig. 1: Model-checking at run-time.

Since our main focus is on non-functional properties, we decided to use *Markov models*, which nicely support quantitative probabilistic systems specifications as well as formalization and verification of requirements, such reliability, performance, and costs, through a *probabilistic temporal logic* and *model checking*. Previous work has shown how model-checking could be used at run-time to support self-adaptation [11], [12], [5], [6]. Through model checking, one can check the desired requirements against the model of the system. Model checking can not only detect a requirement violation but also produces insights into the originating causes. It supports *prediction* of potential violations by applying reasoning on future behaviors that may occur, but may have not occurred yet. It also supports *analysis*

1. http://www.self-adaptive.org

of potential alternatives to reason about competing adaptation strategies. Finally, model checking techniques rely on well-known and effective algorithms that are now available as powerful, off-the-shelf, open-source tools, such as PRISM [13] and MRMC [14].

Traditional model checking techniques and tools, however, in general cannot be applied as they are at run time because they hardly meet the constraints, in terms of performance and memory consumption, imposed by on-line analysis and adaptation. Consider for example the case where run-time verification has to be performed on a node of a sensor network, as in the example we will present in Section 8. Existing model checking tools were in fact conceived for off-line, design-time analysis. This paper discusses how to bring them effectively to run time.

The initial results on run-time efficient probabilistic model checking were presented in [15]. Given a reliability model, knowing which model parameters represent changeable information, and given a set of requirements, the approach statically generates a set of expressions encoding the verification conditions to be evaluated at run time to verify satisfaction of system requirements. In this work models are specified as *Discrete-Time Markov Chains* (DTMCs) and requirements are expressed in *Probabilistic Computation Tree Logic* (PCTL) [16]. This work assumes that changes can be encoded as new values of transition probabilities, modeling events or actions whose probability of occurrence can change at run time. The approach has been extended in [17] to support DTMCs augmented with *rewards*—R-DTMCs [18]—which are more expressive than DTMCs and can also express costs and performance concerns. This paper puts all these findings together in a coherent form, presents a complete formal treatment of the approach and a further extension to also support *sensitivity analysis*. Sensitivity analysis aims at improving the effectiveness of run-time adaptation by identifying the sources of variability that are primarily responsible for requirements violation. As a further contribution, this paper also provides an extended evaluation to assess the applicability of the proposed solution and compare it with other approaches.

This paper is organized as follows. Section 2 provides a detailed problem statement and preview of the approach. Section 3 provides an overview of the mathematical foundations and formalisms upon which the paper is based. Section 4 describes a Web application, which is used throughout the paper to exemplify and validate concepts and techniques. Section 5 and 6 provide a detailed description of the proposed approach for run-time efficient probabilistic model checking. Section 7 motivates and presents sensitivity analysis. Section 8 describes a practical application to the realization of a self-adaptive communication protocol for a wireless sensor network. Section 9 contains a detailed discussion of related work. Section 10 empirically validates the approach also through a comparison with related techniques. Finally, Section 11 contains conclusions and outlines some future work directions.

2 Preview of the approach

As discussed in the previous section, model verification is used at run time to analyze the effect of changes and trigger adaptive reconfigurations. Model checking algorithms, however, are computationally expensive, since they require an exhaustive exploration of the model's state space—which may be very large—to analyze properties that may be arbitrarily complex. The details concerning the complexity of traditional probabilistic model checking can be found in [19], [20], [21]. The computational cost of model-checking may be prohibitive for on-line usage during system operation. Hereafter we give a preview of the approach we devised to make run-time probabilistic model checking efficient.

We focus on systems modeled as Discrete Time Markov Chains (DTMCs), and quantitative probabilistic requirements. DTMCs are a widely accepted formalism to model software system reliability [22], [23], [24]. They are used for design-time reliability assessment of systems composed of interacting parts, such as component-based software or service oriented architectures [25], [26], [27]. DTMCs can be used under the assumption that the system's behavior meets, with some tolerable approximation, the Markov property, i.e. the probability of moving to the next state only depends on the current state, not on the history that lead to that state. This property can be verified as discussed in [28], [29].

As for most design approaches based on Markov Chains (e.g., [29], [30]), we assume that the model describes behaviors that depend on interaction profiles and failure probabilities, which are used

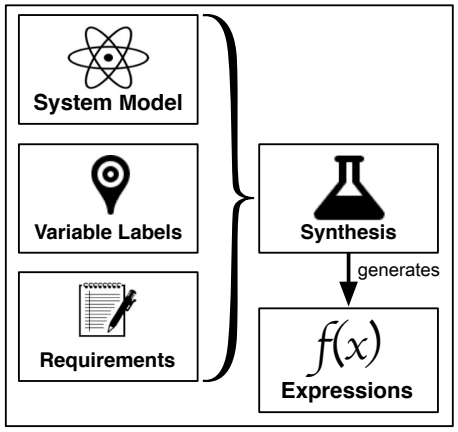
to label transitions with probability distributions. DTMCs can be extended with *rewards*, expressed as numerical labels associated with states or transitions. This extension supports specification of additional cost-related concerns, such as the energy consumption. These concerns may be crucial, for instance in the case of battery-operated devices.

The models developed at design time are often subject to *uncertainty*: certain behaviors are hard to predict, or they may change unpredictably during operation. For example, the failure probability of an operation (represented by a transition in the model) or the energy cost of an operation performed by a device (represented by a state reward) may be hard to predict and estimates gathered by running instances of similar systems may be inaccurate. To account for uncertainty in model parameters, we use *symbolic variables* as labels.

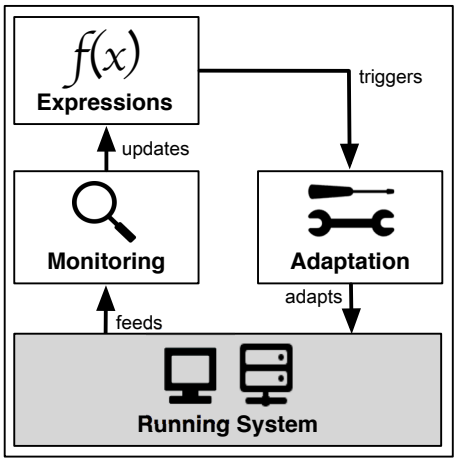
The approach we propose follows two steps, executed at design time and run time, respectively. We refer to the design-time step as *pre-computation* and the run-time step as *verification*, as illustrated in Figures 2(a) and 2(b) . The pre-computation is a *partial evaluation* step [31]. It takes as input: (1) the model of the system in the form of a (R-DTMC), (2) a set of *variable labels*, and (3) the desired system requirements expressed in (R-)PCTL. Variable labels are model parameters whose value becomes known at run time and may change over time. The output produced by the pre-computation step is a partially evaluated set of *symbolic expressions*, which represent a verification condition to be satisfied to meet the requirements. Symbolic expressions are formulae that depend on variable labels and may be evaluated by binding concrete values to them. Evaluation occurs at run time and corresponds to the verification step of the approach.

The actual values to be bound to variables are gathered by monitoring the system in its operational environment. Let us consider an example where a transition in the model leads to a failure state and its variable label represents a component's failure rate, which may change over time. The symbolic verification formula evaluated at design time might represent a requirement that depends on the component's failure rate. The monitor can provide the actual value of the component's failure rate, which can be used to check requirements satisfaction in the current run-time situation.

In short, to achieve the benefits of continuous



(a) Pre-computation.



(b) Verification.

Fig. 2: The two steps of the approach.

on-line verification, instead of executing traditional model checking algorithms, our approach shifts as much of the model verification cost as possible to design time and brings only a residual inexpensive verification step to run time. Precisely, the computationally expensive design-time process that computes symbolic verification conditions reduces run-time model checking to binding variables to the values obtained by the monitor and evaluating the expressions, which is computationally inexpensive and does not require model exploration.

We will present the speed-up obtained by our run-time model checking approach with respect to existing probabilistic model checkers—PRISM [13] and MRMC [14]—and we will also compare it to other competing approaches, pointing out advantages and threats to validity of the different

solutions. Our approach to model checking and the tool we developed to support it are called WM^2. As we will show in Section 10, *WM* 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.

In conclusion, our approach relies on the assumption that, through careful design-time analysis, it is possible to restrict design-time uncertainty and run-time variability to a subset of environment parameters that are modeled as variable model labels. Precisely, we assume that: (1) we can anticipate which variable labels need to be introduced in the model and (2) they are a small fraction of the total number of labels. These assumptions are valid in many practical cases. If the number of varying parameters is large, the approach may still be applied, but would become impractical.

3 Background

This section briefly recalls the necessary background needed to understand the approach we developed to support efficient run-time verification. We first revisit Discrete-Time Markov Chains and then we present the quantitative temporal logic PCTL. Both DTMCs and PCTL are also presented in their extended form with rewards, which enhance the modeling capability. For a more complete treatment of the mathematical model and property language the reader may refer to [19].

3.1 Discrete-Time Markov Chains

A Discrete-Time Markov Chain (DTMC) is a stochastic process satisfying the Markov property and having time domain $T \subseteq \mathbb{N}$. It is defined as a Kripke structure [32] with probabilistic transitions between states. The state space S is here assumed to be finite.

Definition 3.1 (Discrete Time Markov Chain) *A* (labeled) DTMC *is a tuple* (S, s_0, P, L, AP) *where*

- S *is a finite set of states*
- $s_0 \in S$ *is the initial state*
- $P : S \times S \to [0,1]$ *is a stochastic matrix*
- AP *is a set of atomic propositions*
- $L : S \to 2^{AP}$ *is a labeling function that associates to each state the set of atomic propositions that are true in that state.*

In the remaining of the paper, the notation p_{ij} will be used as short form for $P(s_i, s_j)$. An entry p_{ij} represents the probability that the next state of the process will be s_j given that the current state is s_i. Each row i of matrix P is called the *next-state distribution* of state s_i and is formally a categorical distribution [33], implying that the elements in each row sum to one.

The probability of moving from s_i to s_j in exactly two steps can be computed as $\sum_{s_x \in S} p_{ix} \cdot p_{xj}$, that is the sum of the probabilities of all the paths originating in s_i, ending in s_j, and having exactly one intermediate state. The previous sum is, by definition, the entry (i, j) of P^2. Similarly, the probability of reaching s_j from s_i in exactly k steps is the entry (i, j) of matrix P^k. As a natural generalization, matrix $P^0 \equiv I$ represents the probability of moving from state s_i to state s_j in zero steps, i.e. 1 if $s_i = s_j$, 0 otherwise.

A sequence of states $\pi = s_0, s_1, s_2, \ldots$ is an execution *path* through the DTMC if $P(s_i, s_{i+1}) > 0$ holds for any pair (s_i, s_{i+1}). The notation $\pi[i]$ with $i \geq 0$ is used to refer to the i^{th} state in the path π. A path is said to be *finite* if the number of states in the sequence is finite; its length is denoted as $|\pi|$. The probability for a finite path to be traversed is 1 if $|\pi| = 1$, otherwise $\prod_{k=0}^{|\pi|-2} P(s_k, s_{k+1})$. A state s_j is *reachable* from state s_i if there exists a finite path starting in s_i and terminating in s_j.

The states of a DTMC can be classified as either *transient* or *recurrent* [34]. A state s_i is said to be transient iff:

$$\sum_{n=0}^{\infty} P^n(s_i, s_i) < \infty$$

A state s_i is instead recurrent if:

$$\sum_{n=0}^{\infty} P^n(s_i, s_i) = \infty$$

After each visit, a recurrent state will be eventually visited again with probability 1. On the other hand, when the process reaches a transient state there is a non zero probability that it will never

2. The acronym *WM* stands for *working mom*, the name we informally gave to our approach. This comes from the metaphor *cook first, warm-up later* that intuitively describes the distinction between design-time pre-computation and run-time evaluation.

visit it again. Furthermore, in a DTMC the number of visits to a transient state is distributed as a geometric random variable [35]. A recurrent state s_i with $p_{ii} = 1$ is called *absorbing*. If a DTMC contains at least one absorbing state, the DTMC itself is said to be *absorbing*. For simplicity, all the Markov models considered in this work are assumed to satisfy the following property, unless otherwise specified:

> A DTMC model is *well formed* if: (1) all the states are reachable from the initial state, and (2) from every transient state it is possible to reach at least one absorbing state.

Notice that focusing the discussion on well-formed DTMCs does not affect the generality of our technique because it is possible to reduce any of the verification problems we consider in this paper to the probability of reaching an absorbing state in a (equivalent) well-formed DTMC. Similar reductions are common in probabilistic model checking [19], [36] and will be discussed also in Section 5.2.1.

In an absorbing DTMC with r absorbing states and t transient states it is possible to reorder the rows and the columns of the transition matrix P to transform it into the following *canonical form*:

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

where I is an $r \times r$ identity matrix, $\mathbf{0}$ is an $r \times t$ zero matrix, R is a nonzero $t \times r$ matrix and Q is a $t \times t$ matrix.

Since Q specifies only the transitions between transient states, some of its rows sum to strictly less than 1. This is immediate to show for well-formed DTMCs. For the same reason, the following fact holds concerning the probability of absorption (see [37]):

> In a well-formed absorbing Markov chain, the probability of the process to be eventually absorbed is 1 (i.e. $Q^k \to \mathbf{0}$ as $k \to \infty$.).

The number n_{ij} of visits to a transient state s_j, for a process started in s_i, can be computed as the probability of visiting it in the first step, or in the second, or in the third, and so on. In matrix form:

$$N = I + Q^1 + Q^2 + Q^3 + \cdots = \sum_{k=0}^{\infty} Q^k$$

This is a geometric series converging to $(I-Q)^{-1}$. The matrix N is called the *fundamental matrix* of the DTMC.

DTMCs are a valuable formalism to model software execution flows using quantitative data to represent the likelihood of moving from state to state and also to identify failure and success states. Often software modelers need also to express quantitative information concerning some abstract notion of *cost* associated with a state or a transition and a way to quantify accumulated costs as paths are traversed on the DTMC. Concrete examples of costs are the average execution time of a transaction, or its power consumption, or even the monetary cost involved in using a pay-per-use service. To accommodate this modeling requirement, DTMCs may be augmented with *rewards* [18]. Rewards are non-negative real values through which a benefit (or loss) due to the residence in a specific state or the move along a certain transition can be quantified.

Definition 3.2 (Reward-DTMC)
A Reward DTMC (R-DTMC) *is a tuple* (S, s_0, P, L, AP, ρ)*, where S, s_0, P, L, AP are defined as for a DTMC, while* $\rho : S \to \mathbb{R}_{\geq 0}$ *is a* state reward *function assigning a non-negative real number to each state.*

Informally, the total reward cumulated after completing the traversal of a path $\pi = s_0, s_1, s_2, \ldots, s_k$ is $\sum_{i=0}^{i=k} \rho(s_i)$.

According to this definition, rewards are associated with states only. It would be possible to also associate rewards with transitions. It can be proved, however, that an equivalent R-DTMC with only state rewards can always be built given a R-DTMC with both state and transition rewards (see for example [38].)

The definition of a R-DTMC can be generalized by allowing probabilities and rewards to be assigned a symbolic variable instead of a numeric value.

Definition 3.3 (Parametric R-DTMC)
Let Σ_p and Σ_r be two finite sets of symbolic parameters. A parametric R-DTMC *is a tuple* (S, s_0, P, L, AP, ρ)*, where S, s_0, L, AP are defined as for a R-DTMC, while*

- $P : S \times S \to [0,1] \cup \Sigma_p$ *is a parametric stochastic matrix,*
- $\rho : S \to \mathbb{R}_{\geq 0} \cup \Sigma_r$ *is a parametric state reward,*

A *valid evaluation* of a parametric R-DTMC requires the assignments of the symbolic parameters to comply with Definitions 3.1 and 3.2, i.e. 1) each element $\sigma_p \in \Sigma_p$ evaluates to a real value in $[0,1]$ such that for each state the sum of the outgoing transitions is 1, and 2) each element $\sigma_r \in \Sigma_r$ evaluates to a non-negative real number.

Unless otherwise specified, in the remaining of the paper we will consider all R-DTMC to be parametric and all the considered evaluation to be valid.

3.2 Probabilistic Computation Tree Logic

PCTL [16], [39] is a probabilistic branching-time temporal logic, based on the classic CTL logic [19]. A PCTL formula predicates on a state of a Markov process, and evaluates to either *true* or *false*. PCTL may be extended to support rewards yielding R-PCTL [19], [21].

Definition 3.4 (R-PCTL formulae)
R-PCTL formulae are recursively defined by the following syntactic rules:

$$\phi ::= true \mid a \mid \phi \ \wedge \ \phi \mid \neg \ \phi \mid \mathscr{P}_{\bowtie p} \ (\psi) \mid \mathscr{R}_{\bowtie r} \ (\Theta)$$
$$\psi ::= X \ \phi \mid \phi \ U^{\leq t} \ \phi$$
$$\Theta ::= I^{=k} \mid C^{\leq k} \mid \ \Diamond \phi$$

where $p \in [0,1]$, $\bowtie \in \{<, \leq, >, \geq\}$, $t \in \mathbb{N} \cup \{\infty\}$, $r \in \mathbb{R}_{\geq 0}$, *and* $k \in \mathbb{N}$, *and* a *represents an atomic proposition.*

The operator $\mathscr{R}_{\bowtie r} \ (\Theta)$ supports the specification of properties that predicate over rewards. Let us first discuss the semantics of basic PCTL ignoring the reward operator.

Formulae derived from the grammar axiom ϕ are called *state formulae*; those derived from ψ are instead called *path formulae*. Notice that a path formula may only occur as an argument of the probabilistic modal operator $\mathscr{P}_{\bowtie p}(\cdot)$. PCTL differs from its non-probabilistic ancestor CTL since universal and existential path quantification are replaced by the probabilistic operator $\mathscr{P}$.

The temporal operators X and $U^{\leq t}$ are called *Next* and *Bounded Until*, respectively. The *Unbounded Until* can be represented as $U^{\leq \infty}$; this, however, is normally abbreviated as U. As a short and convenient form for *true* $U^{\leq t}\phi$ we can use the derived operator $\Diamond$ (*eventually*): $\Diamond^{\leq t} \ \phi$. It is also customary to abbreviate $\Diamond^{\leq \infty}$ as $\Diamond$.

The semantics of a state formula is defined as follows:

$$\begin{array}{lll} s & \models true & \\ s & \models a & \text{iff} \quad a \in L(s) \\ s & \models \neg\phi & \text{iff} \quad s \not\models \phi \\ s & \models \phi_1 \wedge \phi_2 & \text{iff} \quad s \models \phi_1 \ and \ s \models \phi_2 \\ s & \models \mathscr{P}_{\bowtie p}(\psi) & \text{iff} \quad Pr(\pi \models \psi | \pi[0] = s) \bowtie p \end{array}$$

where $Pr(\pi \models \psi | \pi[0] = s)$ is the probability that a path originating in s satisfies ψ [19].

The following rules define whether a path π originating in s satisfies a path formula ψ:

$$\begin{array}{lll} \pi & \models X\phi & \text{iff} \quad \pi[1] \models \phi \\ \pi & \models \phi_1 U^{\leq t} \phi_2 & \text{iff} \quad \exists 0 \leq j \leq t \quad (\pi[j] \models \phi_2 \wedge \\ & & \quad (\forall 0 \leq k < j \, \pi[k] \models \phi_1)) \end{array}$$

Let us now discuss the reward operator $\mathscr{R}_{\bowtie r} \ (\Theta)$. Intuitively:

- $\mathscr{R}_{\bowtie r}(I^{=k})$ 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 $\bowtie r$.
- $\mathscr{R}_{\bowtie r}(C^{\leq k})$ is true in state s if, from state s, the expected reward *cumulated* after k steps meets the bound $\bowtie r$.
- $\mathscr{R}_{\bowtie r}(\Diamond\phi)$ is true in state s if, from state s, the expected reward cumulated before reaching a state where ϕ holds meets the bound $\bowtie r$.

The third construct can be used, for example, to state the average cost of a run of the system; that is, the expected cumulated cost until the execution reaches a *completion* state.

A more formal definition of the reward fragment semantics can be found in [21]. Intuitively, the expected reward $\mathscr{R}(\Theta)$ for all possible paths exiting a given state s and satisfying the pattern Θ can be computed as the sum of the rewards for each path, weighted by the probability of the path itself (see Section 3.1). The following equations define how the (expected) reward X_Θ over a path π of a R-DTMC is computed for each of the three specification patterns:

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

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

$$X_{\Diamond\phi}(\pi) = \begin{cases} 0 & \text{if } s_0 \models \phi \\ \infty & \text{if } \forall i \; s_i \not\models \phi \\ \sum_{i=0}^{min\{j|s_j \models \phi\}-1} \rho(s_i) & \text{otherwise} \end{cases} \quad (4)$$

A final brief remark concerns the expressiveness of R-PCTL for practical purposes. Once a system is modeled via a R-DTMC, R-PCTL may naturally represent a large number of practically important properties we would like the system to satisfy. For example, the language can express constraints on the probability of reaching an absorbing failure or success state, starting from an initial state. These are examples of *reachability properties*. Reachability properties are expressed by PCTL properties of the kind $\mathscr{P}_{\bowtie p}(\Diamond\ \phi)$, which state that the probability of reaching a state where ϕ holds matches the constraint $\bowtie p$.

4 A Running Example

In this section we illustrate the modeling capabilities of R-DTMCs through an example of a typical Web application, which will be used subsequently to exemplify the proposed approach for run-time efficient probabilistic model checking. The model is shown in Figure 3. It describes a system composed of an HTTP Proxy server, a Web server, and an Application server. Structured data and static content (e.g., files, images, etc.) are stored in a Database and in a File server, respectively. Both of them are cached by ad-hoc cache servers.

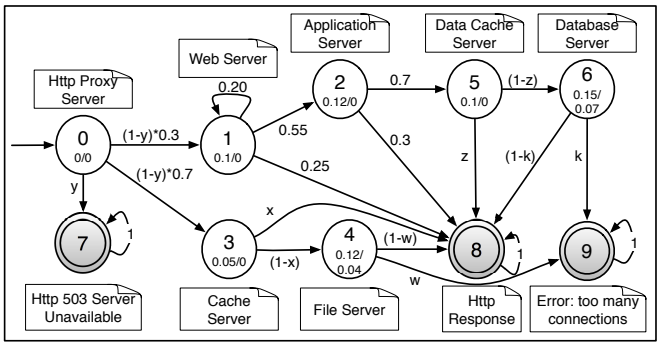


Fig. 3: Example of a parametric DTMC.

In Figure 3 all states are numbered by an integer that uniquely identifies them. With the exception of absorbing states (filled in grey), they represent the execution of a certain operation and are further labeled by a pair in the form n_1/n_2, which represents rewards. Reward n_1 models the *average cost* associated with the operation, while n_2 models the *average latency*. Absorbing state s_7 represents the failure of serving an incoming request due to unavailable server. This models the case where the server is overloaded and drops requests, or the case where it is down because of a maintenance operation. Absorbing state s_9 represents the failure of the execution due to an excessive number of requests to the storage services. Absorbing state s_8 is the endpoint of a correct HTTP request.

Transitions describe the control flow that manages an incoming HTTP request. For example the transitions (s_0, s_1) and (s_0, s_3) are labeled with the probability of the events "a dynamic content has been requested that requires ad-hoc processing" and "a static content has been requested", respectively. Transition (s_1, s_1) corresponds instead to the probability of an HTTP self-redirect. Transitions to absorbing states indicate the final outcome of processing a request.

Parametric transitions indicate that the value of the corresponding probability is unknown, uncertain, or it may change over time. For example transitions (s_3, s_4) and (s_5, s_6) model the cache hit probability, which depends on the current distribution of user requests. Such parameters correspond to the symbolic variables mentioned in Section 2. Notice that, in the example, costs (modeled as rewards) are instead known and fixed and therefore there are no variables to model parametric costs. In matrix form, the model of Figure 3 is characterized by the transient-to-transient (Q) and transient-to-absorbing (R) transition matrices described by Equations (5) and (6):

$$Q = \begin{pmatrix} 0 & (1-y)0.3 & 0 & (1-y)0.7 & 0 & 0 & 0 \\ 0 & 0.2 & 0.55 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.7 & 0 \\ 0 & 0 & 0 & 0 & 1-x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-z \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (5)$$

$$R = \begin{pmatrix} y & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0.3 & 0 \\ 0 & x & 0 \\ 0 & 1-w & w \\ 0 & z & 0 \\ 0 & 1-k & k \end{pmatrix} \quad (6)$$

Table 1 describes a set of requirements for our example, formally specified using R-PCTL. Requirements **R1**-**R4** do not predicate on rewards, while **R5**-**R6** formalize constraints on costs and latencies, modeled as state rewards.

TABLE 1: Requirements **R1**-**R6**.

ID	Informal Definition	PCTL
R1	(*Reliability*): "The probability of successfully handling a request must be at least 0.999"	$\mathscr{P}_{\geq 0.999}(\Diamond\ s = s_8)$
R2	(*Cache hit probability*): "At least 80% of the requests are correctly handled without accessing the database or the file server"	$\mathscr{P}_{\geq 0.8}(\neg(s = s_4) \wedge \neg(s = s_6)\ U\ s = s_8)$
R3	(*Complexity bound*): "At least 70% of the requests must be successfully processed within 5 operations"	$\mathscr{P}_{\geq 0.7}(\Diamond^{\leq 5}\ s = s_8)$
R4	(*Early risk fingering*): "No more than 10% of the runs can reach a state from which the risk of eventually raising an exception is greater than 0.95"	$\mathscr{P}_{\leq 0.1}(\Diamond\ \mathscr{P}_{\geq 0.95}(\ \Diamond s = s_7 \vee s = s_9))$
R5	(*Cost*): "The average cost for handling a request must be less 0.03 dollars"	$\mathscr{R}_{\leq 0.03}(\Diamond\ \ s = s_7 \vee s = s_8 \vee s = s_9)$
R6	(*Response time*): "The average response time must be less than 0.022 seconds"	$\mathscr{R}_{\leq 0.022}(\Diamond\ \ s = s_7 \vee s = s_8 \vee s = s_9)$

5 Run-Time Efficient Verification of PCTL Properties

This section illustrates the algorithms for partial evaluation of PCTL properties at design time, given a system specified as a Markov model. The extension to R-PCTL is presented later in Section 6.

To simplify the discussion, PCTL will be partitioned in several fragments. Section 5.1 deals with *flat*[3] formulae for the reachability of an absorbing state, which have the syntactic form $\mathscr{P}_{\bowtie p}\ (true\ U\ \phi_1)$[4] where ϕ_1 identifies one or more absorbing states. The most common properties expressed in practice can be encoded in PCTL as reachability of an absorbing state [40]. Because of their prevalent usage, and because their treatment is a prerequisite for dealing with other formulae, they deserve a separate description. Section 5.2 addresses the remaining fragments, thus achieving a full coverage of PCTL.

In the discussion we will present both the mathematical methods we devised and their computational complexity. An empirical evaluation of the design-time complexity of partial evaluation will be provided in Section 10.

5.1 Reaching an Absorbing State

Before delving into the mathematical treatment of partial evaluation of flat reachability formulae, we show an example to illustrate the kinds of outputs we generate, referring to the running example introduced in Section 4. Consider requirement **R1** in Table 1. The partial evaluation algorithms defined in this section compute the following parametric expression, which corresponds to the probability of reaching state s_8:

$$\begin{aligned} f_{\mathbf{R1}}(k,w,x,y,z) = &-0.7w - y - 0.144375k + 1 \\ &+ 0.7yw - 0.7yxw + 0.144375zk \\ &+ 0.144375yk + 0.7xw - 0.144375yzk \end{aligned} \tag{7}$$

When a run-time monitor provides the current value of the parameters, the expression (7) can be evaluated and the result can be compared with the threshold 0.999 to verify the satisfaction of requirement **R1**.

Partial evaluation of a flat reachability formula can be performed as follows. 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 as Q) has an inverse N and $N = I + Q + Q^2 + Q^3 + \cdots = \sum_{i=0}^{\infty} Q^i$ [34]. Since an entry q_{ij} of matrix Q represents the probability of moving from the transient state s_i to the transient state s_j in exactly one time step, 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 in state s_i. A Markov process is considered to be *absorbed* when it reaches any of the absorbing states. Notice that $Q^n \to 0$ when $n \to \infty$ (see Section 3.1); hence every well-formed process will always eventually be absorbed, no matter the state it started in.

Whenever the process enters a transient state s_i, its probability of being absorbed in the next time step in the absorbing state s_j is given by the entry

3. A formula is said to be *flat* if none of its sub-formulae contains the $\mathscr{P}_{\bowtie p}(\cdot)$ nor the $\mathscr{R}_{\bowtie r}(\cdot)$ operator.

4. Or equivalently $\mathscr{P}_{\bowtie p}\ (\Diamond\ \phi_1)$ (see Section 3.2).

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 steps), 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 states.

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:

$$Pr(true\ U\ \{s_j \in T\}) = \sum_{s_j \in T} b_{0j} \tag{8}$$

The goal of design-time pre-computation is to compute the value of Equation (8). Matrix B can be computed in different ways, depending on the size of the system and the availability of a parallel or a sequential execution environment.

By definition of matrix product, an entry b_{ij} can be computed as:

$$b_{ij} = \sum_{k=0..t-1} n_{ik} \cdot r_{kj} \tag{9}$$

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

In Sections 5.1.1 and 5.1.2, we present two different approaches for the computation of the entries b_{ik}. The former, based on matrix algebra algorithms, can be quite effective in case of a small number of parameters, even in a sequential execution environment. Furthermore, thanks to its formulation, it is intrinsically parallel and suitable for different kinds of parallelization. The latter instead reduces the problem to the solution of a system of linear equations. This approach is quite efficient for sequential execution environments thanks to the existing implementations of effective heuristics for sparse linear systems.

5.1.1 Matrix-Based Approach

The design-time computation of an entry b_{ij} in general can only be done symbolically, since parametric transitions may be traversed to reach state s_j. The arithmetic complexity of explicitly inverting matrix $I - Q$ of size t, where t is the number of transient states, by means of the Gauss-Jordan elimination algorithm is $O(t^3)$ [41]. The computation of the entry b_{ij} once N has been computed requires $O(t)$ more products, thus the total complexity is $O(t^3 + t) \sim O(t^3)$ algebraic *symbolic* operations on polynomials.

The actual complexity can be significantly reduced if the number c of states having parametric outgoing transitions is small and the transition matrix of the DTMC is sparse, as very frequently happens in practice.

Let $W = I - Q$. The elements of its inverse N are defined as follows:

$$n_{ij} = \frac{1}{det(W)} \cdot \alpha_{ji}(W) \tag{10}$$

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

$$b_{ik} = \sum_{x \in 0..t-1} n_{ix} \cdot r_{xj} = \frac{1}{det(W)} \sum_{x \in 0..t-1} \alpha_{xi}(W) \cdot r_{xj} \tag{11}$$

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 ($\tau \ll n$ by the assumption of sparsity). Determinants can be computed by Laplace expansion. By expanding first the c rows representing the variable states (each has τ symbolic terms), at most τ^c determinants have to be computed and then linearly combined. Each sub-matrix of size $t - c$ does not contain any variable symbol, by construction, thus its determinant can be computed with $(t-c)^3$ operations among numeric values. The latter operation does not involve symbolic terms, hence it is in general much faster. Its actual complexity depends on the precision of floating-point (or rational numbers) representation. On the other hand, memory could easily become an issue for sequential environments because both intermediate results and a possibly large set of sub-matrices have to be stored for processing; for this reason in a sequential environment only small systems can be

analyzed with this algorithm (see Section 10). Thus the final complexity is:

$$O(\tau^c \cdot (t-c)^3) \sim O(\tau^c \cdot t^3) \quad (12)$$

which may significantly reduce the original complexity assuming $\tau \ll n$ (sparsity) and c small (few symbolic states), since here we execute $\tau^c \cdot t^3$ arithmetic operations on numeric values instead of symbolic expressions. This 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 [19], which has complexity $O(n^3)$ [42] in a sequential computational model.

Notice that the procedure described in this section is naturally parallelizable in several ways. First, the sum in Equation (11) is intuitively formalizable by a map-reduce pattern [43], where the *map* operation is the computation of each cofactor and the *reduce* performs the weighted sum according to the coefficients r_{xj}. Furthermore, since the cofactor of a matrix containing symbolic entries can be computed by Laplace expansion, it is possible to design a hierarchical map-reduce configuration. This approach is valid both for multicore and distributed execution environments. The main limitation in case of multicore could be the amount of memory required to store all the intermediate results. Second, parallel algorithms for matrix algebra have been largely studied, mostly for numeric application, allowing for efficient computation of at least the numeric cofactors in (11) [44].

5.1.2 Equations-Based Approach

In this section we present an alternative approach to symbolic reachability computation that does not rely on the possible speedup of parallel execution. As we observed, very often the transition matrix of the DTMC modeling a software system is 1) very sparse, since each component typically interacts with a limited number of other components, and 2) presents some regular topological patterns, which reflect an implicit design rationale. By formulating the computation of the elements b_{ij} as the solution of a linear system of equations, it is possible to exploit state-of-the-art heuristics and provide a significant speed-up in the actual execution time.

Let us first observe that the inverse of a (non-singular) square matrix A satisfies the following property: $A \cdot A^{-1} = I$. Hence, the i-th column of the matrix A^{-1} corresponds to the solution the following system of linear equations:

$$A \cdot v = e_i \quad (13)$$

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 to solve Equation (9) one needs to compute the entries of the i-th row of the matrix $N = (I-Q)^{-1}$, it is possible to exploit a property of the transpose of invertible matrices, namely $(A^{-1})^T = (A^T)^{-1}$. The i-th row of $(I-Q)^{-1}$ corresponds 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 aforementioned property. The problem of calculating the row of the matrix N and, through (9), of B is thus reduced to the solution of a linear system of equations.

Let us now evaluate the computational complexity of this approach. Solving linear equation systems is a well studied mathematical problem, even though most of the available libraries concern numerical solutions and cannot deal with symbolic parameters [45]. The most popular algorithms to solve linear equation systems embedded in probabilistic model-checkers are iterative ([46], [47]). They can efficiently solve even large systems with the desired precision of the final result and without requiring a large amount of memory.

In our case it is not possible to adopt the same strategies because iterative methods do not deal conveniently with symbolic parameters. The presence of unknown parameters makes it hard to assess the convergence of the solution. For this reason *direct* method have been adopted, optimized for the solution of sparse linear systems [48].

In [17] we presented an approach that is supported by a solver based on structured Gaussian elimination and Markowitz pivoting [48]. Structured Gaussian elimination is a variation of the widely used method to triangularize linear systems, which reduces the solution of a large sparse equation system to the solution of a small dense one. This reduction step 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 are 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*. To reduce fill-in during the elimination steps, we adopted Markovitz pivoting to select the next element to be eliminated. Other strategies can be more efficient for specific cases but their discussion is beyond the scope of this paper. The interested reader may refer for example to [48].

To avoid any loss of accuracy during intermediate computation steps, our design-time solver uses infinite precision rational numbers for all the numeric values in the models. All the mathematical procedures have been implemented in Maple 15[5].

5.2 Towards Coverage of Full PCTL

Although many practically relevant requirements may be specified as the reachability of an absorbing state [40], there are cases where the properties of interest require the expressive power of full PCTL. In this section, algorithms will be provided to extend the approach to handling all the remaining fragments of PCTL.

We will start by discussing flat unbounded Until formulae, whose pattern is $\mathscr{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$. Being flat, neither ϕ_1 nor ϕ_2 nor their sub-formulae can contain the operator $\mathscr{P}_{\bowtie p}(\cdot)$. This class is indeed a superclass of the fragment $\mathscr{P}_{\bowtie p}(\Diamond\phi)$ studied in the previous section. Afterwards, in Section 5.2.3, we will present algorithms to verify the bounded operators X and $U^{\leq t}$, concluding the verification of the flat fragment. Formulae with nested $\mathscr{P}_{\bowtie p}(\cdot)$ operators will be treated in Section 5.2.4.

5.2.1 Flat Until Formulae

An example of a flat Until formula is given by requirement **R2** in Table 1 for the running example introduced in Section 4. This formula specifies that the process is required to reach state s_8 without traversing states s_4 and s_6.

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 we have already seen. This reduction process goes through the following transformation of the DTMC model.

5. http://www.maplesoft.com

Given a DTMC $D=(S,s_0,P,L)$ and a flat until formula $\mathscr{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$, a DTMC $\bar{D}$ is derived from D through the following procedure:

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 introduce a single transition toward s_{goal} with probability 1.
3) For all the states where $\neg(\phi_1 \vee \phi_2)$ holds, remove all the outgoing transitions and introduce a single transition toward s_{stop} labeled with probability 1.

The state space of $\bar{D}$ is $S\cup\{s_{\text{goal}},s_{\text{stop}}\}$; the labeling function of $\bar{D}$ is extended accordingly by the two atomic predicates S_{goal} and S_{stop} holding only in states s_{goal} and s_{stop} respectively. The transition matrix of $\bar{D}$ will be denoted as $\bar{P}$.

Theorem 5.1 (Flat Until Verification) *$\mathscr{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$ holds in state s_i of D iff $\mathscr{P}_{\bowtie p}\ (\Diamond\ s_{goal})$ holds in state s_i of $\bar{D}$.*

Proof 5.1 *For a path π of D originating in s_i and satisfying the path formula $\phi_1 U \phi_2$ there exists $k\geq 0$ such that $\pi[k]\models\phi_2$ and for all j, $0\leq j<k\ :\ \pi[j]\models\phi_1\wedge\neg\phi_2$. By construction, there will exist one and only one path $\bar{\pi}$ in $\bar{D}$ such that for all j, $0\leq j\leq k\ :\ \bar{\pi}[j]\equiv\pi[j]$ and $\bar{\pi}[k+1]\models S_{goal}$. Furthermore, $Pr(\pi)=Pr(\bar{\pi})$ because, by construction, for all j, $0\leq j<k\ P(\pi[j],\pi[j+1])=\bar{P}(\bar{\pi}[j],\bar{\pi}[j+1])$ and $\bar{P}(k,s_{goal})=1$.*

By virtue of Theorem 5.1, verification of $\mathscr{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$ can be reduced to verification of the flat reachability property $\mathscr{P}_{\bowtie p}\ (\Diamond\ s_{\text{goal}})$ on $\bar{D}$, for which the algorithms defined in Section 5.1 hold.

Let us consider again the example of requirement **R2** in Table 1 for the example of Section 4. According to the construction procedure discussed here, in the derived DTMC $\bar{D}$ state s_8 will be connected with probability 1 to s_{goal}, while states s_4 and s_6 are connected with probability 1 to state s_{stop}. Satisfaction of requirement **R2** can be evaluated by computing the probability of reaching the absorbing state s_{goal}, which results in the following symbolic expression:

$$\begin{aligned} f_{\mathbf{R2}}(k,w,x,y,z) &= 0.7x-0.155625y \\ &\quad +0.144375z-0.144375yz \\ &\quad -0.7yx+0.155625 \end{aligned} \tag{14}$$

Notice that the parameters k and w do not appear in the expression, meaning that heir value is irrelevant for the verification of **R2**, as it can be intuitively assessed by looking at the DTMC of Figure 3.

5.2.2 Alternative formulations for classes of flat Until properties

Before proceeding, in this section we report alternative direct algorithms for two classes of flat until formulae, which do not require the construction of the transformed process $\bar{D}$. The former is based on the so-called *first-step* analysis [49]. The latter is based on an algorithm to compute reachability of a transient state from the theory of stochastic processes [34].

First-Step Analysis for Flat Until

First-step analysis [49] exploits the locality properties of Markov processes to transform the problem of computing flat until formulae into the solution of a system of linear equations, structurally different from the one presented in Section 5.1.2 but still yielding the same solution. First-step analysis will also be exploited in Section 6 to formalize the verification of reward properties.

First-step analysis for flat until formulae $\mathscr{P}_{\bowtie p}\ (\phi_1\ U\ \phi_2)$ requires solving the following system of linear equations:

$$u^*(s_i) = \sum_{s_k \in S} p_{ik} \cdot u^*(s_k) \tag{15}$$

subject to the following boundary conditions:

$$u^*(s_i) = \begin{cases} 1 & \text{if } s_i \models \phi_2 \\ 0 & \text{if } s_i \models \neg(\phi_1 \vee \phi_2) \\ 0 & \text{if } s_i \in C \ \wedge\ s_i \not\models \phi_2 \end{cases} \tag{16}$$

where C is the set of absorbing states of D.

The boundary conditions (16) describe the elementary cases for the computation of u^*. The first and second case present an immediate correspondence to steps 2 and 3 of the construction of the previously defined $\bar{D}$. The third case accounts for the possible presence of absorbing states satisfying ϕ_1 but not ϕ_2, thus not included in the second case; from such states there is clearly no chance to satisfy $\phi_1 U \phi_2$ because of their absorbing nature.

The solution of the system of linear equations defined by (15) and (16) involves symbolic computations in presence of model parameters. The algorithms defined in Section 5.1.2 obviously apply also to this case.

Reachability of Transient States

An alternative procedure is presented here to compute a special case of the flat until property: $true U \phi$, where ϕ identifies transient states. Reachability of transient states can be computed without transforming a DTMC D into a DTMC $\bar{D}$, based on the theoretical background from [34].

First, the probability of reaching a transient state from an absorbing state is trivially 0, while the probability of reaching s_j from itself is trivially 1. For any two distinct transient states s_i and s_j, let f_{ij}^k be the probability that the first hitting of state s_j happens at time k, given that the process started from s_i and let X_k represent the random variable representing the state of the process at time k:

$$\begin{cases} f_{ij}^0 = 0 \\ f_{ij}^k = Pr(X_k = s_j \wedge \forall 0 \leq y < k\ X_y \neq s_j | X_0 = s_i) \end{cases} \tag{17}$$

Let

$$f_{ij} = \sum_{k=0}^{\infty} f_{ij}^k \tag{18}$$

Thus, f_{ij} represents the probability of ever reaching state s_j given that the process started from s_i. Notice that, for every well-formed DTMC (see Section 3.1) $f_{0j} > 0$, because every state of the model has to be reachable from the initial state. Furthermore, since the only recurrent states are the absorbing ones, $f_{ii} < 1$ for every the transient state s_i.

Though Definition (18) formalizes the probability of reaching a transient state, the computation of its actual value is not straightforward from the definition. This can instead be done by recalling the definition given in Section 3.1 of the fundamental matrix N, whose entries n_{ij} represent the expected number of visits to the transient state s_j before absorption, given that the process started in s_i. Assuming the values f_{ij} to be known, n_{ij} can be derived by conditioning on whether state s_j is ever visited:

$$\begin{aligned} n_{ij} = &\ E(\text{number of visits to state } s_j \mid X_0 = s_i) \\ = &\ n_{jj} \cdot f_{ij} \end{aligned} \tag{19}$$

In other words, the value n_{jj} is the expected number of "returning" visits to s_j given that it is eventually

reached from state s_i (see [34], page 190). From Equation (19), it is immediate to derive:

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

Summing up, $s_i \models \mathscr{P}_{\bowtie p}\ (\Diamond\ \ s = s_j)$ iff $f_{ij} \bowtie p$. This relationship allows for the use of the matrix-based algorithms provided in Section 5.1.1.

5.2.3 Flat Bounded Formulae

In a flat bounded state PCTL formula the arguments of the $\mathscr{P}_{\bowtie p}(\cdot)$ operator is a flat Next or a Bounded Until path formula.

Let us first consider the Next operator. The set of paths to consider in order to estimate the probability of a path formula $X\phi$ in a state s_i is the set of all the 1-step long paths originating in s_i. Since ϕ is flat, the states satisfying ϕ can be identified once for all at design time. The transition matrix P contains the probability of moving from a state to another in a single step. Hence, computing the probability of reaching, from a state s_i, a state where ϕ holds in 1 step, can be computed as:

$$Pr(X\ \phi) = \sum_{s_j \models \phi} p_{ij} \tag{20}$$

As for the Bounded Until operator, notice that each path originating in a state 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. Referring to a DTMC $\bar{D}$ constructed as in Section 5.2.1, each of these paths corresponds to a path in $\bar{D}$ that exactly at step $k+1$ reaches the state s_{goal}. Hence, any path of D satisfying $\phi_1 U^{\leq t} \phi_2$ corresponds to a path in $\bar{D}$ being at step $t+1$ in state s_{goal}.

The probability distribution of the states reached after exactly $t+1$ time steps in $\bar{D}$ can be computed by raising the transition matrix $\bar{P}$ to the power of $(t+1)$:

$$Pr(X_k \models \phi_2 \wedge k \leq t \mid X_0 = s_i) = \bar{P}^{t+1}(s_i, s_{\text{goal}}) \tag{21}$$

Summarizing, $s_i \models \mathscr{P}_{\bowtie p}\ (\phi_1 U^{\leq t} \phi_2)$ on D iff $\bar{P}^{t+1}(s_i, s_{\text{goal}}) \bowtie p$ on $\bar{D}$.

As an example of the evaluation of a flat Bounded Until, consider requirement **R3** in Table 1 for the running example illustrated in Section 4. The resulting parametric expression corresponding to the probability of reaching state s_8 within 5 steps is given by Equation (22).

$$\begin{aligned} f_{\mathbf{R3}}(k,w,x,y,z) = {} & 0.10548 - 0.10548y \\ & + (0.0231 - 0.0231y)z \\ & + (0.165 - 0.165y)(0.7 - 0.7z)(1-k) \\ & + (0.165 - 0.165y)(0.3 + 0.7z) \\ & + (0.7 - 0.7y)(1-x)(1-w) \\ & + (0.7 - 0.7y)x \end{aligned} \tag{22}$$

5.2.4 Nested Formulae

The analysis of PCTL has been so far restricted to its flat fragment, where the arguments of a $\mathscr{P}_{\bowtie p}(\cdot)$ 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 defined in the previous sections.

In the case of *nested* formulae, some information needed to compute the desired parametric expression may only become available at runtime. For instance, consider requirement **R4** in Table 1 for the running example introduced in Section 4: the set of states from which an error will eventually occur with probability greater than .95 will only be known at run time, because it depends on the actual value of the model parameters. Thus, the probability of reaching any of these states cannot be computed at design time with the previously defined procedures, because it would not be possible to identify the target states. To evaluate a formula with nested $\mathscr{P}_{\bowtie p}\ (\cdot)$ operators, we need to know in which states its sub-formulae are satisfied. The same consideration can be applied recursively to sub-formulae of a sub-formula, until a flat one is reached that can be directly analyzed.

To deal with this issue without losing the benefits of parametric verification, the solver needs to delay at run time the evaluation of a nested formula, until all the knowledge concerning its sub-formulae has been gathered.

Focusing on Until formulae, the solution provided in Section 5.2.1 is based on the construction of the modified DTMC $\bar{D}$. Such a construction requires certain states to be disconnected from their successors and then connect them to either s_{goal} or s_{stop}.

As previously explained, the resulting parametric expression would be computed as the reachability of the absorbing state s_{goal} in $\bar{D}$.

To delay the decision about the connection of a state to s_{goal} or to s_{stop} at run time, we simply need to add three more parameters to each state. The first is a coefficient α_i that multiplies all the elements p_{ij} of D. The second and the third are parameters $\beta_{i\ \text{goal}}$ and $\beta_{i\ \text{stop}}$, respectively, that define $\bar{P}(s_i, s_{\text{goal}})$ and $\bar{P}(s_i, s_{\text{stop}})$. The three additional parameters can assume either 0 or 1 as a value, and their intuitive purpose is the following: by assigning 0 to a parameter α_i state s_i is disconnected from all its successors; by assigning 1 to either $\beta_{i\ \text{goal}}$ or $\beta_{i\ \text{stop}}$ state s_i becomes connected to state s_{goal} or s_{stop}, respectively. Notice that, because of their meaning with respect to the definition of DTMC, for every valid parameter assignment one and only one among α_i, $\beta_{i\ \text{goal}}$, and $\beta_{i\ \text{stop}}$ can take the value 1.

Computing the probability of a path $\Diamond\ s_{\text{goal}}$ at design time leads to a parametric expression having as variables both the model parameters and the additional parameters α_i, $\beta_{i\ \text{goal}}$, and $\beta_{i\ \text{stop}}$ for each state s_i. At run time, 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 $\bar{D}$.

The recursive application of this procedure on nested formulae keeps the benefits of parametric analysis, though it requires as many evaluations as the nesting depth of a formula. Assuming that most nested formulae used in practice have limited nesting levels, the impact on complexity would still be limited. Another drawback for run-time analysis of nested formulae is that the resulting mathematical expressions are in general longer than in the case of flat formulae due to the presence of more parameters, but the evaluation time would still be much faster than the execution of a conventional model-checking routine for systems of a realistic size. An evaluation of the impact the number of parameters has on the design-time complexity will be provided in Section 10.

At design time, the computation of Next and Bounded Until nested formulae follows the same principle just described for Until, and they have to be computed on the model instrumented with the additional parameters α_i, $\beta_{i\ \text{goal}}$, and $\beta_{i\ \text{stop}}$. The adaptation of the mathematical procedure for the Next operator is straightforward. The main issue with this approach is the computational complexity at design time. Indeed, the additional parameters may have a high impact on the execution time of the algorithms from Section 5.1. To leverage this issue, parallel implementations may be used on high performance platforms, or, for not too large systems, the results for each combination of the α and β parameters can be stored in a direct access table. Notice though that the number of entries of such table would be $O(3^{|S|})$ and the size of each of them would be up to $O(|S|^{\log(|S)})$ because all the transitions in the model would become symbolic.

6 Verification of R-PCTL

This section extends the partial evaluation approach to also cover rewards. Equations (2), (3), and (4) in Section 3.2 formalize the semantics of the three specification patterns used to express reward-related properties.

Some of the mathematical procedures presented in this section are based on the notion of *expected reward* along a set of paths originating from a state s_i. In Section 3.2 this value has been intuitively defined as the sum of the rewards cumulated along each of the paths, weighted by the probability for that path to be taken. Since such a sum may contain infinite terms, it could be unfeasible to compute it directly from its definition.

Applying first-step analysis (Section 5.2.2), the expected reward for a (non empty) path originating in s_i can be computed by the following linear equation:

$$r_i = \rho(s_i) + \sum_{s_j \in S} p_{ij} \cdot r_j \tag{23}$$

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

Notice from Equation (23) that if s_j is an absorbing state and its state reward $\rho(s_j)$ is strictly positive the equation has no finite solutions. Indeed, in such a situation, for all the states from which s_j is reachable the expected reward would be infinite. For this reason, we assume here that $\rho(s_j) = 0$ for all the absorbing states s_j.

The following Section 6.1 discusses the verification of unbounded formulae of the class $\mathscr{R}_{\bowtie r}\ (\Diamond\phi)$, while Section 6.2 will deal with the bounded operators $I^{=k}$ and $C^{\leq k}$.

6.1 Unbounded Formulae

A formula $\mathscr{R}_{\bowtie r}\ (\Diamond\phi)$ is true in a state s_i if the expected cumulated reward before reaching a state satisfying ϕ meets the constraint $\bowtie r$. To simplify the exposition of an algorithm that computes it, we will only discuss flat R-PCTL formulae, meaning that in path formulae $\Diamond\phi$, ϕ can only be a Boolean combination of atomic propositions. The extension to the nested fragment of R-PCTL can be achieved by instrumenting the R-DTMC with additional parameters according to the procedure defined in Section 5.2.4 for nested PCTL formulae.

The expected cumulated reward over all the paths satisfying $\Diamond\phi$ and originating in a state s_i can be computed by constraining the linear system (23) to the following boundary conditions:

$$r_i = \begin{cases} 0 & \text{if } s_i \models \phi \\ \infty & \text{if } s_i \in C \wedge s_i \not\models \phi \end{cases} \tag{24}$$

where $C \subseteq S$ is the set of absorbing states of the R-DTMC.

The rationale behind the boundary condition (24) is intuitive: a state s_i satisfying ϕ is the last state of a path π that satisfies 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 indicates a path that will never satisfy $\Diamond\phi$ and thus contributes to the accumulation of rewards as an infinite cost, according to definition (4).

The solution of (23) subject to (24) is a rational polynomial expression whose unknowns are the model parameters used to label transition probabilities and state rewards. For the algorithms to solve this system of equations and their complexity, the reader may refer to Section 5.1.2. Summing up, $s_i \models \mathscr{R}_{\bowtie r}\ (\Diamond\phi)$ iff $r_i \bowtie r$.

As an example of partial evaluation of an unbounded R-PCTL formula, let us consider requirement **R6** in Table 1 for the running example introduced in Section 4. Partial evaluation yields the following expression:

$$\begin{aligned} X_{\Diamond(7\leq s\leq 9)} &= 0.21734375 + 0.084yx - 0.084x \\ &\quad - 0.21734375y - 0.02165625z \\ &\quad + 0.02165625yz \end{aligned} \tag{25}$$

Before concluding this section, let us discuss the special case of the cumulated reward before absorption. This special case corresponds to the computation of the expected cost of a run, regardless of its termination in a success or failure state. Its computation can be pursued either by modifying the boundary conditions of Equation (24), imposing $r_i = 0$ for all the absorbing states ($s_i \in C$), or by means of the following matrix algebraic procedure.

Assuming that for all the absorbing states $s_i \in C$ $\rho(s_i) = 0$, a reward can only be accumulated by the visits to transient states. From Section 3.1, an entry n_{ij} of the fundamental matrix N represents the expected number of visits to the transient state s_j before absorption, given that the process started in s_i. Since after each visit to s_j the reward $\rho(s_j)$ is gained, let ρ be a column vector with elements $[\rho(s_0), \rho(s_1), \rho(s_2), \dots]$ and $C \subset S$ the set of absorbing states; the expected cumulated reward before absorption can be computed by the following equation:

$$X_{\Diamond(s_i \in C)} = N \cdot \rho \tag{26}$$

Equation (26) is equivalent to (23) subject to the boundary conditions (24) modified as previously indicated:

$$r = \rho + Q \cdot r$$

$$(I - Q) \cdot r = \rho$$

$$r = (I - Q)^{-1} \cdot \rho$$

This equivalence between the matrix algebraic formulation and the linear equation system (23) allows the choice of the most convenient solution algorithms for the execution environment at hand, as will be discussed later in Section 10.

6.2 Bounded Formulae

A formula $\mathscr{R}_{\bowtie r}\ (I^{=k})$ is true in a state s_i if the expected state reward at time k meets the bound $\bowtie r$. By definition, the expected value of the cumulated reward can be computed as the sum of the rewards associated with every state reachable in exactly k time steps, weighted by the probability of reaching it. The probability of reaching a state s_j from a state s_i in exactly k time steps is the entry (s_i, s_j) of the matrix P^k (see Section 3.1).

In a more compact way, let ρ be a column vector with elements $[\rho(s_0), \rho(s_1), \rho(s_2), \dots]$. The

expected reward $X^{=k}$ can be computed by the following equation:

$$X_{I^{=k}} = P^k \cdot \rho \tag{27}$$

where an element $X_{I^{=k}}[i]$ corresponds to the expected reward from state s_i. Hence, $s_i \models \mathscr{R}_{\bowtie r}\ (I^{=k})$ iff $X_{I^{=k}}[i] \bowtie r$.

A formula $\mathscr{R}_{\bowtie r}\ (C^{\leq k})$ is instead true in a state s_i if the expected reward cumulated after k time steps satisfies the constraint $\bowtie r$. For the previous considerations, the expected reward gained at the j-th step is exactly $P^j \cdot \rho$. Thus, to compute the cumulated reward up to the k-th step with $k \geq 1$ it is possible to apply the following equation:

$$X_{C^{\leq k}} = \sum_{j=0}^{k-1} P^j \cdot \rho \tag{28}$$

When $k = 0$, $X_{C^{\leq k}} = 0$ by definition (3). Hence, for all $k > 0$, $s_i \models \mathscr{R}_{\bowtie v}\ (C^{\leq k})$ iff $X_{C^{\leq k}}[i] \bowtie v$.

As a final remark, notice that despite having a logarithmic complexity in the order of the exponent k, computing a large power of a matrix may be computationally heavy, both in terms of time and memory. On the other hand specific efficient solutions have been introduced for equations of the form of (27) and (28), such as [50, p. 121]. Matrix power can also be easily parallelized, e.g., decomposing the matrix into blocks [45]. Alternatively, the standard graph-based algorithms used by probabilistic model-checkers for bounded properties can be straightforwardly adapted to deal with parametric probabilities and rewards [19].

7 Sensitivity Analysis

Sections 5 and 6 described how to compute parametric closed formulae at design time that can be used for an efficient run-time evaluation of system properties, when the values of unknown parameters become known. Interestingly, parametric formulae also support a powerful reasoning tool called *sensitivity analysis*. In this section we introduce, define and exemplify sensitivity analysis for our context referring, where necessary, to the running example illustrated in Section 4.

Intuitively, sensitivity analysis is the process of assessing how the value of a certain system property is affected by changes in the values of its constituents. This assessment may contribute to different goals, such as:

- **Support to decision making**: when limited resources are available for system improvement (whether in terms of money or time) sensitivity analysis may indicate the critical parts of the system under analysis that have a more significant impact on the satisfaction of system requirements;
- **Guidance to improvements**: errors are more likely to lead to global failures if they occur in crucial parts of the system. Sensitivity analysis can assign a degree of importance to each part and guide designers towards a more detailed and focused analysis;
- **Maximization/Minimization of properties**: when optimizing with respect to a certain metric of interest, sensitivity analysis may be used to find values of system parameters that maximize or minimize the metric under analysis (see Section 8);

Sensitivity analysis may be performed at design time as well as at run time. At design time engineers may use it to improve the design of the system. They may exploit it to make informed decisions among competing design choices. Alternatively, at run time, the information gathered through sensitivity analysis may support self-adaptation by prioritizing an appropriate adaptation action among multiple possible adaptation alternatives.

Sensitivity can be formalized as follows:

Definition 7.1 (Sensitivity) *Let* $\omega = \{\omega_0, \omega_1, \ldots, \omega_n\}$ *be the set of parameters of a Markov model D,* $f : \omega \to \mathbb{R}$ *a function of* ω *(corresponding to a quantitative property of D). The sensitivity S of* f *with respect to* ω *is defined as:*

$$S(f, \omega) = (\vec{\nabla} f)_\omega = \left[\frac{\partial f}{\partial \omega_0}, \frac{\partial f}{\partial \omega_1}, \ldots, \frac{\partial f}{\partial \omega_n} \right]$$

For a given set of operating conditions $\overline{\omega} = \{\overline{\omega_0}, \overline{\omega_1}, \ldots, \overline{\omega_n}\}$ (i.e., the assignment of specific values to the parameters of D), the partial derivative $\partial f / \partial \omega_i$ evaluated in $\overline{\omega}$ yields an insight into the way f changes when ω_i changes. More precisely, we build a closed-form expression for sensitivity which helps in identifying dangerous or desirable scenarios

over all the parameters space. For example, it is possible to rank the parameters ω_i according to the impact they have on f to understand which parameter most significantly affects the satisfaction of the system requirement represented by f.

An effective sensitivity analysis builds on top of two fundamental assumptions. First, sensitivity is just an indicator. Indeed, it does not account for feasibility of a change. Second, the proposed approach for sensitivity analysis is based on the assumption that the parameters ω_i are independent of each other. This requires special care in modeling.

Let us consider as practical example requirement **R1** of our running application. By computing the partial derivative of Equation (7) with respect to the set of transition variables $\omega = \{w, z, x, k, y\}$ we obtain the following sensitivity formulae:

$$\begin{aligned}
\partial f/\partial w &= -0.7 + 0.7y - 0.7yx + 0.7x \\
\partial f/\partial z &= +0.144375k - 0.144375yk \\
\partial f/\partial x &= -0.7yw + 0.7w \\
\partial f/\partial k &= -0.144375 + 0.144375z \\
&\quad +0.144375y - 0.144375yz \\
\partial f/\partial y &= -1 + 0.7w - 0.7xw \\
&\quad +0.144375k - 0.144375zk
\end{aligned}$$

By considering the following set of operating conditions:

$$\overline{\omega} = \{w = 0.05, z = 0.3, x = 0.35, k = 0.05, y = 0.01\}$$

we obtain the following results:

$$\begin{aligned}
\partial f/\partial w &= -0.45045 \\
\partial f/\partial z &= 0.0071465625 \\
\partial f/\partial x &= 0.03465 \\
\partial f/\partial k &= -0.100051875 \\
\partial f/\partial y &= -0.972196875
\end{aligned}$$

These results clearly indicate that, in these operating conditions, a change of variable y's value affects the reliability of the system (i.e., requirement **R1**) more significantly than changes to the other variables. In the context of our example, this suggests that, to increase the fault tolerance of the application, we have to prioritize the actions that affect the availability of the Web server. For instance, we may decide to deploy more instances of this component. In a similar way, sensitivity analysis may be applied to other system requirements. For example, the sensitivity of the system's response time (requirement **R6**) with respect to its parameters may be evaluated by studying Equation (25).

The computed value of sensitivity directly depends on the set of selected operating conditions (i.e., the point in which the derivative is computed). This dependence may be considered as a weakness of sensitivity analysis. Indeed, a system with an extremely variable behavior may yield totally different sensitivity results in different operating conditions. However, this variability can be easily quantified with Taylor's theorem [51] that may be used as a measure of confidence on the computed sensitivity results.

Finally, it is important to notice that the sensitivity formulae we derive in our approach can also be used for *reverse* sensitivity analysis, i.e. the identification of operational conditions that are particularly sensitive to specific parameters. Such information could allow a designer to devise mechanisms that keep the system in a "safe" operational region, where its properties are less sensitive to uncertainty or variability of external parameters.

8 The Low-power Wireless Bus case study

This section illustrates the application of the proposed approach to a real-world embedded software system from the Wireless Sensor Networks (WSNs) [52] domain. The case study demonstrates the need for a run-time efficient probabilistic verification approach to support an implementation of a WSN system that uses the communication protocol Low-power Wireless Bus (LWB) [53]. The next paragraphs are organized as follows: first we briefly illustrate the LWB protocol, second we introduce a R-DTMC model for an LWB node, and finally we apply our approach pointing out its applicability and practical advantages.

8.1 Description

LWB is a recently proposed WSN communication protocol that turns a multi-hop low-power wireless network into an infrastructure similar to a shared bus, where all nodes are potential receivers of all data. It achieves this by mapping all traffic demands on a type of fast *network floods* (i.e., transmissions of packets from an originator node to all other nodes in the network). As a result,

LWB inherently supports one-to-many, many-to-one, and many-to-many traffic. LWB also keeps no topology-dependent state, making it more resilient to link changes due to interference, node failures, and mobility than prior approaches. More precisely, as illustrated in [53], LWB maps all communication on fast Glossy floods [54]. A single flood serves to send a packet from one node to all other nodes. To avoid collisions between floods, LWB uses a time-triggered operation: nodes communicate according to a global communication schedule that determines when a node is allowed to initiate a flood.

The protocol operates within communication rounds that repeat with a *round period* T computed at the host (i.e., a sensor node acting as leader of the network) and based on the current traffic demands. Every round consists of a number of non-overlapping communication slots. In each slot, at most one node puts a message on the bus (i.e., initiates a Glossy flood), whereas all other nodes read the message from the bus (receive and relay the flood). A round starts and ends with a slot allocated by the host to distribute the communication schedule. Thus each round contains, for redundancy reasons, two copies of the schedule. The schedule includes the round period T and the mapping of individual nodes to the following data slots. The detailed description of the protocol is beyond the scope of this paper and can be found in [53].

The efficiency of WSN applications significantly depends on the efficiency of their communication protocol. The configuration parameters of each communication protocol have to be tuned according to the application demands and operating conditions. In particular, in this specific class of systems, *energy efficiency* is the main aspect that drives parameter configuration. In the case of LWB, application designers may choose among different values for the period T in the allowed range: $1000\text{ms} \leq T \leq 60000\text{ms}$.

LWB has a startup phase, which synchronizes each node with the LWB host and occurs when a new node joins the network and each time a node misses more that three consecutive schedules. Since this phase may have severe impact on energy consumption, the protocol designer carefully tries to optimize its behavior. Different values of parameter T lead to significantly different performance in terms of consumed energy. In addition, an a-priori choice of T at design-time is not necessarily the most appropriate strategy to achieve an energy efficient startup synchronization since the most convenient value depends on the specific execution conditions. Values of T close to the maximum are in general appropriate (i.e., they provide a lower overall energy consumption). However, high values of T also imply an increased amount of energy required for the startup node synchronization. The trade-off between high and low values of T directly depends on the reliability of communications among nodes. An a-priori choice made at design time would not be a good solution. A seamless and continuous run-time adaptation of value T would instead be necessary.

TABLE 2: Cost/reward structures for the LWB model in Figure 4. Each reward represents the energy consumption expressed as the worst case radio on time in ms.

State	Energy Consumption
B_e	(T-1000)
B_b	1000
R_e	35
R_b	145
S_e	19

8.2 Modelling LWB nodes

The choice of an appropriate value for period T that minimizes the energy required for the startup synchronization of LWB applications can be performed by exploiting a probabilistic model that captures the energy consumption of the nodes. Figure 4 shows the parametric R-DTMC of an LWB node as described in [55]. The variable label p_s in this model represents the probability for the node to receive a schedule that was sent by the host. Justifying the assumptions and accuracy of this model is beyond the scope of our paper; these details can be found in [55].

Starting from the R-DTMC model in Figure 4 we can extract an absorbing R-DTMC that focuses only on the states involved in the initial synchronization. This second R-DTMC, reported in Figure 4, is also augmented with the cost/reward structure detailed in Table 2. The reward structure associates each state with the (worst case) time in milliseconds for which the LWB node has the radio switched on.

As we said, the goal is to choose high values of T, but also define an upper bound to constrain

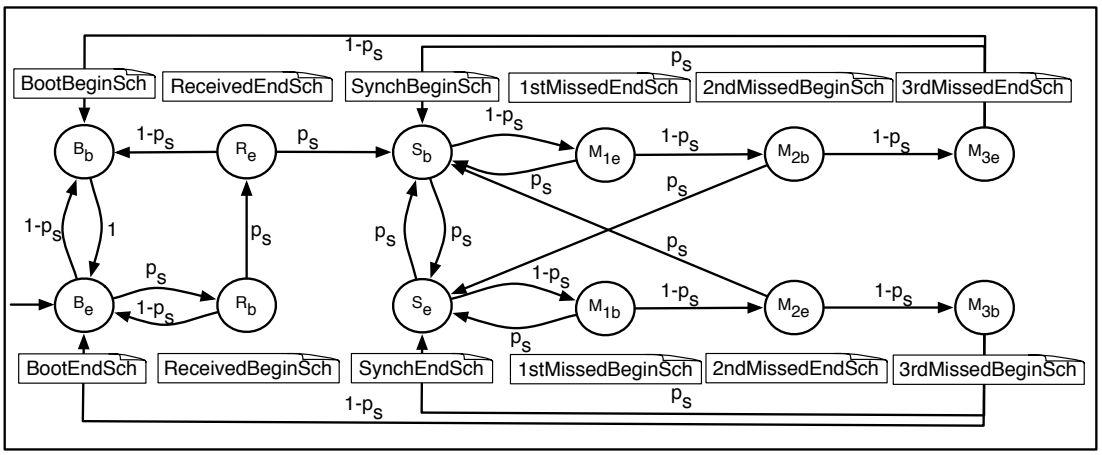


Fig. 4: Parametric Markov chain model of an LWB node, taken from [55]

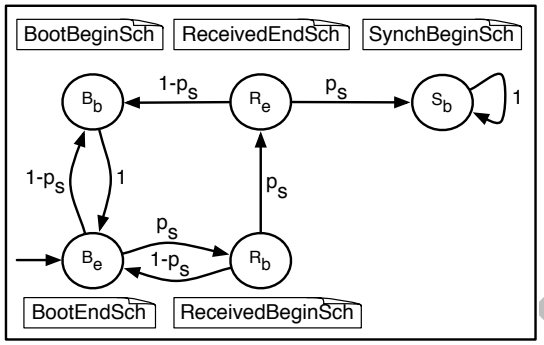


Fig. 5: Absorbing Parametric Markov chain model of the startup synchronisation of an LWB node

the energy consumption in the startup phase. The following requirement $\mathbf{R}_{LWB}$ expresses the goal: *The startup radio on-time should be as close as possible, but less than 10s*. The value of the radio on-time period can be queried on the model at design time using, for example, the PRISM model checker after setting a value for p_s. The query to submit is $\mathscr{R} =?(\Diamond s = s_b)$. By choosing different values for p_s, a designer can explore how different values of T produce distinct startup energy consumptions and can choose an appropriate period T given an expected value for p_s. It is easy to realize that some values for T are appropriate in some operating conditions but invalid in others. For example considering an expected p_s equal to 0.8 and setting T to $5600s$ yields a worst case radio on-time of $9.9s$ that satisfies requirement $\mathbf{R}_{LWB}$. In a scenario of less reliable communications (i.e., $p_s = 0.6$) the same value of T produces instead a worst case radio on-time equal to $24s$, which clearly violates the requirement. Thus, the choice of T may be conveniently postponed to a run-time adaptive algorithm that adapts to the changing operating conditions.

To compute at run-time the most appropriate value for T we need to periodically evaluate requirement $\mathbf{R}_{LWB}$ considering the current value of p_s. Deploying a probabilistic model checking engine on a WSN node is however out of question, since nodes are severely constrained in terms of storage and computational power. As an example, the ultra low-power wireless sensor modules TMote Sky[6] are equipped with a 8MHz microcontroller, 10k RAM memory, and 48k of flash memory. In addition, the energy saved by the run-time adaptation of T would be compensated by the energy spent to periodically run the model checking engine. Furthermore, WSN applications typically have strict timing requirements to self-adapt, which would not be compatible with traditional probabilistic verification techniques. The WM approach comes into play to overcome these obstacles.

The WM pre-computation step of $\mathscr{R} =?(\Diamond s = s_b)$ produces the following symbolic expression:

$$f_{\mathbf{R}_{LWB}}(T, p_s) = \frac{-1000p_s^3 + 1035p_s^2 - 855p_s + T}{p_s^3} \tag{29}$$

The WM verification step is a fast and inexpensive computation that can be easily performed by a sensor node. Given a value for p_s estimated at

6. http://www.crew-project.eu/sites/default/files/tmote-sky-datasheet.pdf

run-time[7], the LWB host can compute the symbolic expression with respect to the desired requirement (i.e., $f_{\mathbf{R}_{LWB}}(T, p_s) < 10.000$), obtaining the desired value for period T. In the case of the TMote Skyby sensor modules formula (29) can be evaluated in less than 1ms.

9 Related Work

Quantitative verification at run time is necessarily subject to strict time constraints [11]. Although verification procedures can be efficient enough for certain application domains [56], [12], traditional techniques were conceived for design-time analysis and therefore they are usually not acceptable in terms of time consumption [57].

Improving the efficiency of the current model checkers has long been one of the goals of the research community. Some approaches have been proposed to tackle the specific issues of run-time analysis. In this section, we discuss related work by grouping contributions in three classes: *incremental analysis*, *parameter space exploration*, and *parametric model checking*.

9.1 Incremental Verification

Incremental approaches are in general characterized by two steps: (1) localize the impact of a change in the artifact, and (2) reuse saved results from previous analyses to avoid unnecessary re-computation.

Incremental analysis approaches for non-probabilistic systems have been proposed to improve the generation or the exploration of the state space of the model by identifying which previous results are still valid after a change and which have to be re-computed [58], [59], [60], [61], [62], [63].

Only few papers have been published on incremental quantitative verification of probabilistic models; e.g.,[64], [65]. This work focuses on discrete-time Markov Decision Processes (MDPs)[8], a superset of R-DTMCs, and reachability properties. In the target usage scenario for this technique, the model has to be re-analyzed after (a few) transition probabilities change. As a first step, the Markov model is partitioned into its maximal strongly connected components (SCCs). Roughly speaking, SCCs can be analyzed in isolation and then the local results can be combined to obtain the probability of reaching the target states. When a change occurs, the set of the SCCs that are directly affected is generated. Then, a search algorithm is applied to identify all the SCCs indirectly involved. This search algorithm is based on a convenient topological order of the SCCs. Specifically, let C be an SCC and let $Pre^*(C) \subseteq S - C$ be the set of states from which C is reachable; let us also assume that any of the target states is reachable from C. A change in the transition probabilities included in C may affect the probability of its predecessors to reach the target, but not its successors. This observation supports an efficient search strategy that goes through a model's SCCs and re-analyzes only those which may have been affected by the change. Furthermore, this analysis procedure is also parallelizable, thanks to the partial ordering among the SCCs: at any step an SCC can be processed independently of the others as long as its successors have been analyzed. An explicit application to run-time verification of this approach has been presented in [66][9].

Compared to our approach, [64], [65] provides a higher flexibility on the structure of the DTMC, since virtually any change is allowed. On the other hand, the benefits of incrementality heavily depend on the topology of the system and the localization of a change, providing no guarantee on the required verification time. Moreover, it does not support nested formulae.

The approach introduced in [68], Δ *evaluation* for reliability analysis, is concerned with incremental reliability analysis based on conveniently structured DTMCs. The structure of these models follows the proposal of [28], where each software module (represented by a state of the DTMC) can either transfer control to another module, fail by making a transition toward an absorbing failure state, or complete the execution by moving toward an absorbing success state. Assuming that a single entry of the transient-to-transient transition sub-matrix Q changes, there is no need to re-compute the reliability of the entire system from scratch, but a few arithmetic operations can be used to correct

7. The host easily estimates p_s by exploiting the sequence number attached to messages; see [55] for more details.

8. An MDP can be roughly seen as R-DTMC augmented with the possibility of non deterministic transitions [19].

9. This approach also supports a speedup in the first analysis, as shown in[67]

the previous reliability value. Despite its efficiency, Δ evaluation can only deal with a single change at a time in the matrix Q and does not provide support neither for generic DTMC models nor for general PCTL properties, but only for the reachability of a specific absorbing state [10]. Compared to our approach, [68] requires a special structure of the DTMC, tailored to reliability analysis. In terms of computation time, it saves the design-time burden of computing the parametric formulae but does not provide substantial improvements at run time.

Finally, [69] reports some preliminary results of a novel approach to incremental quantitative analysis of software artifacts. The approach is based on the syntactic structure of the artifact being analyzed and the verification procedure is formalized via an attribute grammar. Both parsing and attribute evaluation after each change are managed by incremental algorithms. This approach has been applied to incremental reliability analysis of workflow languages [70], and in [38] for more general quantitative analysis of structured languages. In principle, it provides higher flexibility, allowing not only parameters but also the model structure to change at run time. However, there is still no evidence that the method would lead to an improved efficiency for specific classes of models (e.g., Markov processes) and properties (e-g-, PCTL).

9.2 Parameter Space Exploration

The approaches described in this section reverse the perspective of verification: instead of checking if a parametric Markov model satisfies a PCTL property ϕ, they try to synthesize the set of parameter evaluations that make the model satisfy ϕ. Though these techniques were not explicitly designed for run-time analysis, their application is straightforward since at design time they can in principle explore the whole parameter space and store in a convenient lookup table all the evaluations that make the model satisfy ϕ. This way, when a change occurs at run time, a quick access to the lookup table can provide an immediate answer to the verification problem.

In [71], given a parametric MDP and an evaluation of the model parameters, the procedure resolves all the non deterministic choices to their optimal combination, i.e., one that maximizes the probability of reaching a set of target states for that evaluation. This combination is called an *optimal schedule*. When the optimal schedule is found for a specific evaluation, the parameter space is explored starting from the given evaluation until the maximum bounded region is found for which the scheduler is still optimal.

The approach introduced in [72] can instead deal with the entire PCTL, still verified on MDPs. The parameter space is divided into hyper-rectangles such that all the elements of a hyper-rectangle either satisfy the desired property or they refute it. The approach is iterative and keeps partitioning the search space until a minimum size for the regions is reached. Since it is in general impossible to cover the parameter space by hyper-rectangles, a part of it may remain undecided. Hence the verification procedure is not complete.

Being MDP models (augmented with a reward structure) a superset of R-DTMCs (the latter having only one possible action per each state), the previous techniques can also be applied to R-DTMCs. Despite their generality, however, parameter space exploration approaches suffer from some limitations compared to our approach. First, the complexity of exploring the parameter space is in general exponential in the number of parameters, though several heuristics can be applied to speed it up in practice (e.g., [73]). This can make design-time computation unfeasible, even for models with few parameters. Second, the storage of a large number of parameter regions corresponding to the satisfaction of a certain property can be an issue.

9.3 Parametric Model Checking

The class of parametric model checking approaches includes our approach and is essentially based on the pre-computation at design time of a closed-form expression corresponding to the satisfaction condition of the desired quantitative property. The first approach for parametric model-checking of DTMCs has been proposed by C. Daws in [74]. The main contribution of this seminal work is the synthesis of parametric closed formulae through a *state elimination algorithm*, similar to the one used in automata theory to synthesize regular expressions from finite state automata [75]. The same algorithms has been previously introduced in the field of stochastic control theory, though with a different perspective (see e.g., [76] pg. 114).

10. Notice that at least two absorbing states are required for the approach to work correctly [68]).

More precisely, Daws' algorithm computes a closed stochastic expression corresponding to a flat reachability property. The length of the expression has been proved to be, in the worst case, $O(n^{log(n)})$ (n being the number of states) [74]. In its original formulation, Daws did not provide support for rewards. An efficient implementation of Daws' algorithm presented in [77], [78], [79] combines state space reduction techniques and early evaluation of the regular expression to improve the actual execution time when only few variable parameters appear in the model. The improvement in [78] requires n^3 algebraic operations among polynomials, performing better than [74] in most practical cases, although still leading to an $O(n^{log(n)})$ long expression in the worst case. The approach in [78] has been implemented in the tool called PARAM. This tool also supports verification of flat reward properties. Concerning expressiveness, the approach we describe in this paper provides support for the full PCTL, while Daws' algorithm only deals with reachability. PARAM could represent an alternative to our approach for the verification of reachability properties. For this reason a deep comparison between the efficiency of the two tools on the flat reachability fragment will be reported in Section 10.

9.4 Sensitivity Analysis

Sensitivity analysis has been performed according to Definition 7.1 in several research contributions. In [28], an analytical procedure is provided for DTMC-based reliability analysis of component-based systems. Reliability is formalized as the probability of reaching an absorbing "success" state of the DTMC. Sensitivity is then computed with respect to the failure probability of each component, by algebraic operations on the transition matrix. A similar approach is presented in [80], where the sensitivity is computed not only for reliability but also for response time. In [81] and [82], reliability analysis is extended to also deal with error propagation among components, and sensitivity of system's reliability is computed with respect to the probabilities that a component experiences an unrecoverable failure or introduces an error in the data-flow that is propagated to other components. Besides sensitivity analysis, perturbation analysis can be used to quantify the impact of applying small variations to a model parameter on the satisfaction of a global property. Perturbation analysis is mostly performed through Monte Carlo simulations. While established in several engineering disciplines, it has been recently applied to Software Engineering, e.g., in [83]. By relying on our approach, which generates closed-form expressions corresponding to quantitative properties, the computation of sensitivity is significantly more efficient at run time.

10 Empirical Evaluation

This section discusses a set of experiments conducted to empirically assess the design-time effort required by the approach described in this paper, which has been implemented by the *WM* toolset. To provide a basis for comparison, the same test cases have been also analyzed through the PARAM model checker [79], which can be considered as the direct competitor for reachability formulae.

All the test cases have been generated randomly and are well-formed models. The algorithm used for the generation is available online[11], as well as the full data set for the experiments. Each test case in the data set is identified by the seed used to initialize the random generator, to make all the experiments replicable.

Each test case has exactly two absorbing states and five outgoing transitions from each state. The properties analyzed in the tests are only of the forms $\mathscr{P}_{=?}(\diamond\phi)$ and $\mathscr{R}_{=?}(\diamond\phi)$, where ϕ uniquely identify one of the absorbing states. These types of properties have been chosen for two reasons: first, they stress the core routines for all the unbounded properties (see Sections 5 and 6); second, they can be analyzed also by PARAM, which does not support the full R-PCTL.

In the comparison, we used PARAM version 1.8, which runs on 64 bit processors. The binary distribution has been downloaded from the official website[12]. By default, PARAM uses a bisimulation preprocessing to reduce the size of the input model [84]. Since the focus of the comparison is on the verification algorithms only, and considering that the same preprocessing can be applied in our case too, bisimulation has been disabled by using the "–no-lump" flag on command line invocation of PARAM.

11. http://www.antonio.filieri.name/publications/preprints/2015-tse/

12. http://depend.cs.uni-saarland.de/tools/param

The execution time for PARAM is reported as measured by the tool itself, launched with the statistics flag enabled.

The WM toolset consists of a Java preprocessor that takes as input the DTMC model and the target states to be reached and produces a Maple code implementing the algorithms defined in the previous sections. The expression computed in Maple can then be directly exported to either Java or C code. The execution time of the Maple implementations of the WM algorithms has been measured using the *time[real]()* built-in function to record the start and the end of the execution, according to the official directives of the tool. The execution time reported here does not include the set-up of the tools, which is anyway independent of the specific instances and negligible with respect to the analysis time. The execution environment for all the experiments is a Dual Intel(R) Xeon(R) CPU E5530 @ 2.40GHz with 8 Gb of ram, running GNU Linux Ubuntu Server 11.04 64bit. All the tests reported in this section did not overrun the available memory.

The first set of experiments analyzes the performance of the matrix-based algorithms with respect to the number of states and the number of model parameters. The results are reported in Section 10.1 and compared with PARAM. The second set of experiments analyzes the design-time efficiency of the equation based algorithms with respect to the same problem dimensions, by comparing the WM implementation both with PARAM and with the built-in solver of Maple 15; the results are discussed in Section 10.2. In Section 10.3, a set of experiments has been conducted to stress the current implementation of the WM and to provide an empirical assessment of its execution time with larger models. Finally, in Section 10.4 runtime performance of the WM approach is compared with well-established tools for probabilistic verification, to provide evidence of the run-time speedup.

10.1 Matrix-Based Algorithms

Matrix-based algorithms enjoy the benefit of splitting symbolic and numeric computation. The former, being based on Laplace expansion, has an exponential complexity in the number of parametric states. After the expansion, the determinants of a large number of numeric sub-matrices have to be computed. While established numeric solutions can scale to very large sub-matrices [45], storing these intermediate sub-problems is memory demanding, especially for sequential computation environments. Parallelization can be leveraged to increase scalability, but this is out of the scope of this evaluation. For these reasons we limit the size of the test models to 30 states, as well as to 30 symbolic states, to make the experiments feasible within the memory bound of the execution environment, which is sequential. For every configuration pair *(number of states, number of parameters)*, we analyze 50 randomly generated cases.

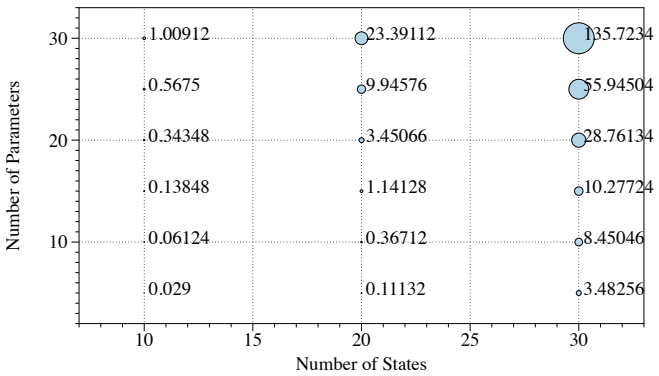


Fig. 6: Matrix based approach: average execution time (s) vs number of states / number of parameters.

Fig. 7: Matrix based approach: maximum execution time (s) vs number of states / number of parameters.

Figure 6 reports the average execution time for the test suite, while Figure 7 reports the maximum execution time. The choice of reporting both the average and the maximum execution time is due to the high variance of the results. Indeed, model topology affects the actual execution time in way that is hard to predict. Recalling the algorithm of Section 5.1.1, specific topologies may reduce the number of non zero cofactors, speeding up the actual execution

time. This phenomenon is intuitively more relevant in the case of a large number of model parameters, because Laplace expansion of the parametric rows leads to smaller numerical cofactors; due to the sparsity of the matrix, such small cofactors are more likely to be trivially singular (having determinant equal to zero), and this condition is efficiently identified by Maple. On the other hand, the relation between the average and the maximum execution time seems to be essentially linear. In particular, the worst cases take at most twice the time of the average cases. All the samples have been analyzed in reasonable time (<6 minutes), though the largest ones almost saturate the available memory (<8Gb). The growth of execution time is quite regular, with respect to both the number of states and the number of parameters.

The theoretical complexity described by Equation (12) is empirically verified in Figure 8. The x-axis corresponds to the complexity index $O1 = \tau^c \cdot (n-c)^3$ computed for each sample set (τ is the average number of transitions originating in a state, set in our experiments to 5; n is the number of states of the model; c is the number of states having at least one parametric outgoing transition).

The polynomial fitting of the empirical data with respect to the complexity index $O1$ yields the following relation:

$$17.686 + 1.4706 \cdot 10^{-6} \cdot O1$$

with a correlation between the estimated mode and the data set of 0.93103 and the determination index R^2 equal to 0.86681[13].

The verification of the same sample cases with PARAM was not always possible because the execution time of the model checker when the models contain more than 10 parameters drastically increases. Considering the execution time of the matrix-based algorithms reported above, all the runs of PARAM taking more than 5 hours have been interrupted. For this reason Figures 9 and 10 have the y-axis truncated at 10.

By looking at Figures 9 and 10, three observations can be made. First, the execution time of

13. In statistics, R^2 is a quality index for models whose main purpose is the prediction of future outcomes on the basis, in this case, of the measured execution times. For its use in this section, $R^2 \in [0,1]$ and a perfect fitting between them model and the data would correspond to $R^2 = 1$.

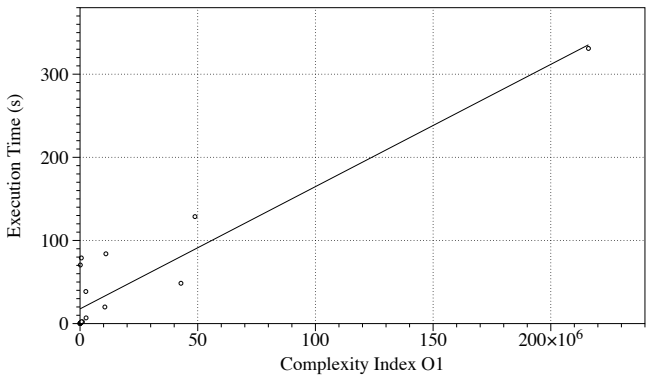


Fig. 8: Matrix based approach: empirical validation of the complexity assessment in (12).

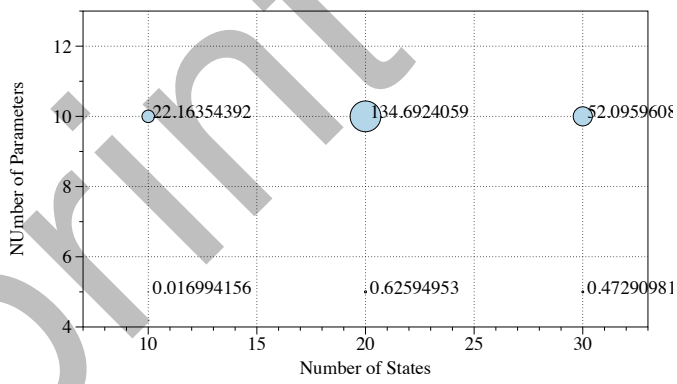


Fig. 9: PARAM: average execution time (s) vs number of states / number of parameters.

the matrix-based approach is significantly lower than the execution time of PARAM, both in the average and in the worst case. Second, there is a higher variability in the execution times of PARAM verifications, as evidenced by the ratio between the maximum and average value for each sample set (up to 360). Third, there is a monotonic trend of the execution time of PARAM with respect to the number of parameters, but this is not the case for the number of states, at least in the average case (see Figure 9).

10.2 Equation-Based Algorithms

In this section, the equation-based algorithms defined for the WM approach are compared with both PARAM and the solution of the linear equation systems by means of the built-in solver provided by Maple 15. Due to the high variability observed in the result sets of the three analyzers, the plots in this section report both the average execution time,

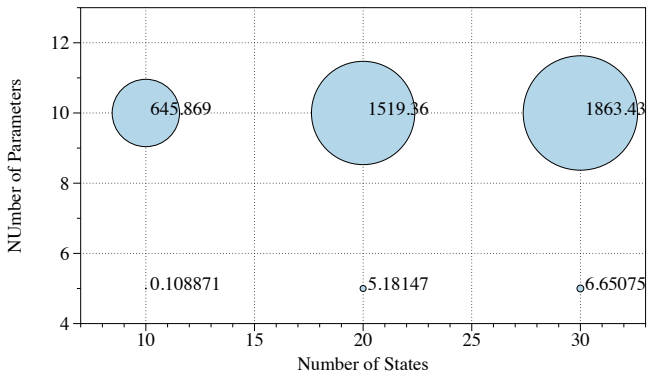


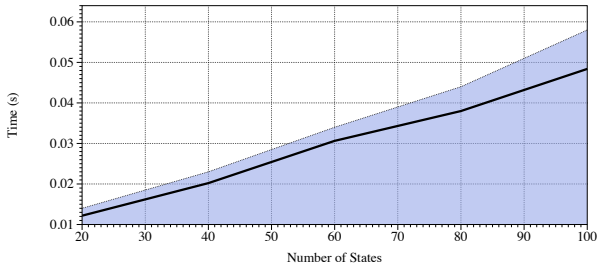
Fig. 10: PARAM: maximum execution time (s) vs number of states / number of parameters.

as a thick black line, and the maximum measured execution time, as a dashed thin line.

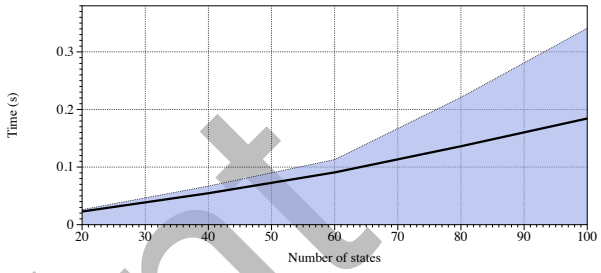
Figure 11 refers to the reachability of an absorbing state. All the input models have exactly 5 parametric transitions. Each data point represents the average and maximum execution time of 25 samples, respectively. The three tools provide reasonable performance, though the WM and the Maple built-in solver are quite faster. There is also a regular monotonic trend in the performance of the equation-based procedures, while PARAM presents a higher variability. The minimum value of the average execution time curve of PARAM, not easily readable in the scale of Figure 11(c), is $0.13s$, significantly larger than the other two approaches for the same input size.

Figure 12 shows again the results of analysis of the same reachability property, but for random input models having exactly 10 parametric transitions. As before, 25 samples per point have been analyzed. In this test suite the performance of the three tools is no longer comparable, with the equation based solvers running in seconds, while PARAM takes up to 5 hours (with minimum value for the average execution time curve of $137s$, not easily readable in the figure's scale). Furthermore, for models larger than 80 states (shadowed in Figure 12), PARAM always takes more than 5 hours and the corresponding records have been discarded; this is the reason for the 0 execution time reported on the graphs.

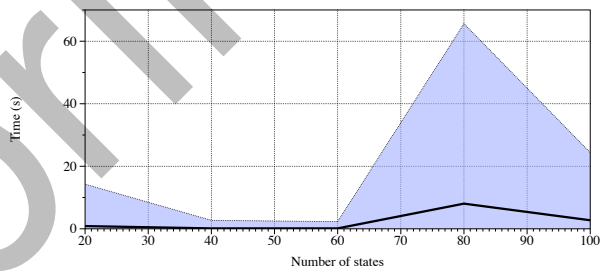
Figure 13 shows the execution time of the three solvers analyzing the property $\mathscr{R}_{=?}(\Diamond\phi)$, again with respect to the number of states of the input models. Each input model has exactly 7 parameters, 2 of which are parametric rewards; 1 absorbing state;



(a) Equation based algorithms of WM.



(b) Maple built-in solver.

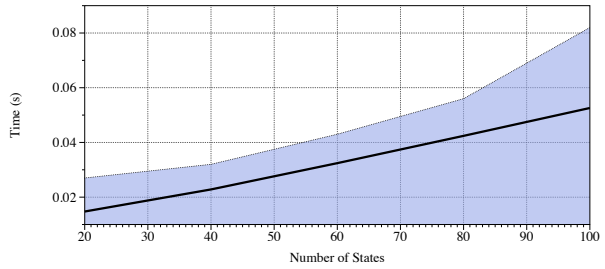


(c) PARAM.

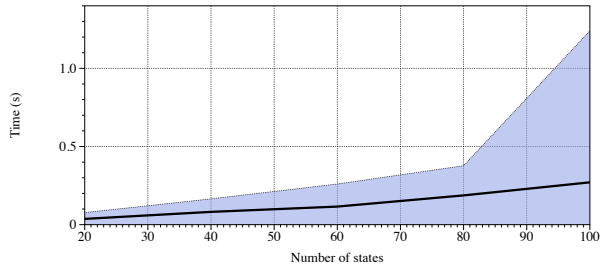
Fig. 11: Execution time vs number of states: flat reachability, 5 parametric transitions (the thin dashed line represents the maximum execution time while the think continuous line the average execution time).

5 outgoing transitions for each transient state. For each input size, 50 samples have been taken.

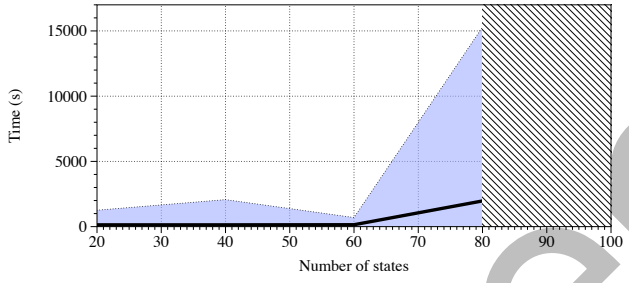
Figures 11, 12, and 13 show the difference in performance of the three solvers. In particular, the plotted experimental data indicate that WM and the Maple built-in solver outperform PARAM. With 100 states and 10 parameters PARAM takes almost always more than 5 hours to verify the reachability property. The equation based solvers can perform the same tasks in tens of seconds. On the other hand, no significant differences can be noted between the Maple built-in solver and the WM implementation at this level of complexity of the input models. In the next section the two equation based procedures will be stressed with more complex models in order



(a) Equation based algorithms of WM.

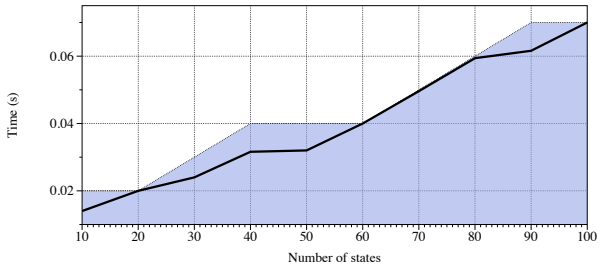


(b) Maple built-in solver.

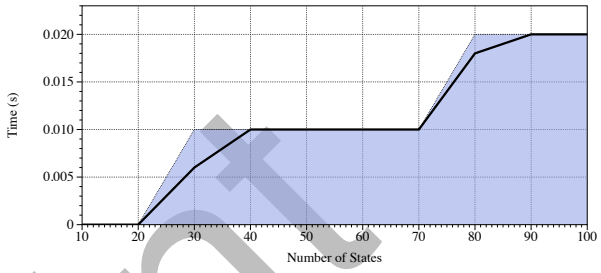


(c) PARAM.

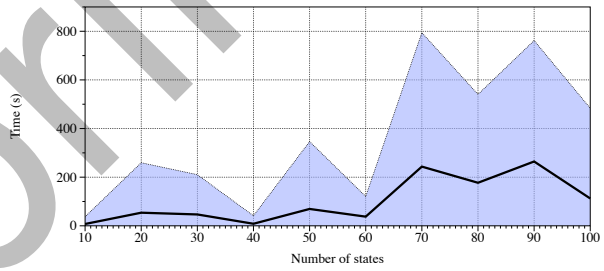
Fig. 12: Execution time vs number of states: flat reachability, 10 parametric transitions (the thin dashed line represents the maximum execution time while the think continuous line the average execution time).



(a) Equation based algorithms of WM.



(b) Maple built-in solver.



(c) PARAM.

Fig. 13: Execution time vs number of states: cumulative reward, 5 parametric transitions, 2 parametric rewards.

to show the benefits of the WM implementation.

10.3 Empirical Complexity of the WM Implementation

In this section a set of complex input models are analyzed to compare the performance of the Maple 15 built-in solver and the WM implementation.

Figures 14 and 15 show the results of analysis with the two equation-based solvers for a set of input models with 100 states and 5 outgoing transitions per state. The property under analysis is a flat reachability formula. Each point of the plots corresponds to 100 samples. The WM implementation significantly outperforms the built-in solver of Maple: the former never took more than 5 minutes, while the latter ran for more than 4 hours in the worst case. Hence, the fill-in reduction strategies adopted for the WM implementation proved to be effective in speeding up partial evaluation. The two figures also show that though the average execution time is monotonically growing, the maximum execution time does not have a regular trend. This issue is evident for the larger models, where the impact of topology is not negligible.

We made additional experiments to further stress the WM implementation and empirically assess the actual performance of the tool (which is notably less than the one expected from the worst case analysis of Section 5.1.2). Figure 16 shows the results of analysis for a flat reachability formula and a set of input models with 200 states and 5

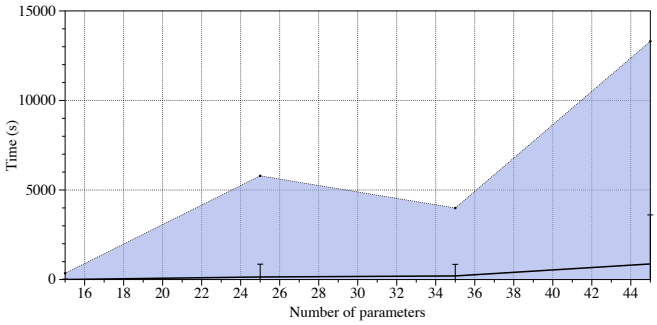


Fig. 14: Stress test of the Maple built-in solver: 100 states, up to 45 transition parameters.

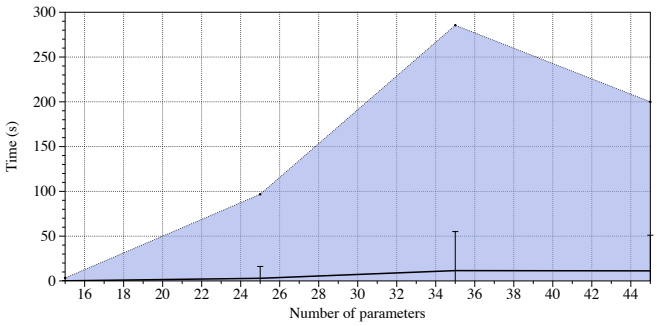


Fig. 15: Stress test of the WM implementation: 100 states, up to 45 transition parameters.

outgoing transitions per state. Each point of the plots corresponds to 25 samples.

10.4 Runtime Performance

Before concluding this section, Figure 17 provides a quantitative glimpse of the benefits achievable at run time by using the WM approach (the verification time is reported in logarithmic scale).

The evaluation time of the closed-form expressions produced by WM is reported in comparison with two popular model-checkers, PRISM [85] v4.0.2 and MRMC [14] v1.4.1.Concerning the execution time of PRISM, only the model-checking time is reported, as measured by the tool itself; the model construction time is not considered [85]. All the tools have been required to produce results with an accuracy of at least 10^{-15} and have been executed with their default configuration.The execution environment is the same as described for the previous experiments.

The test suite is composed by 128 randomly generated DTMCs, having 50 to 500 states (with step 50), two absorbing states, and a normally distributed number of outgoing transitions per state with mean 10 and standard deviation 2. The number of variable states is 4 in each model, thus the number of parameters of each model is normally distributed with mean 40 and standard deviation 8. The formulae computed in Maple for the WM have been encoded into C functions and compiled[14]. The property to verify is reachability of one of the two absorbing states.

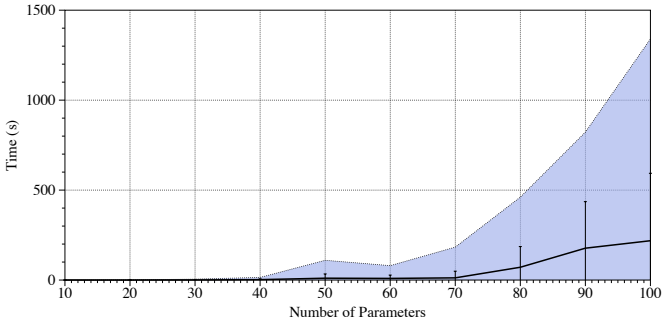


Fig. 16: Stress test of the WM implementation with 200 states and up to 100 parameters.

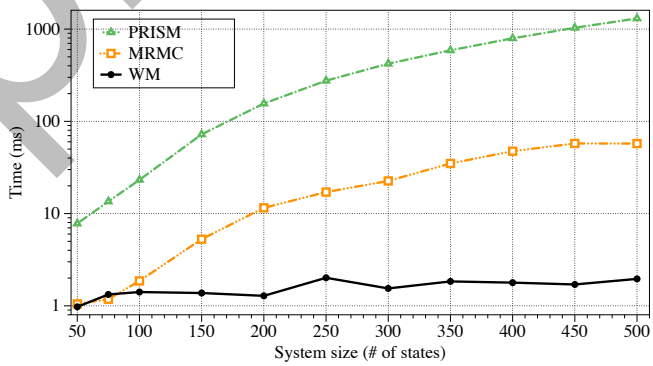


Fig. 17: Runtime performance of the WM compared with MRMC and PRISM.

A thorough discussion of the performance time of each of the tools is out of the scope of this paper, only a few observations follow concerning the issues of bringing those verification tools at run time.

Both PRISM and MRMC show a fast-growing empirical complexity with the size of the model. PRISM seems slower than MRMC. This is most likely due to the nature of the models and the property at hand; indeed the PRISM symbolic engine provides a higher speedup in presence of more

14. C has been preferred to Java because the latter has a limitation in the size of a method's body that could create problems with long mathematical expressions.

complex properties and larger models [85]. On the other hand, MRMC, thanks to its fast explicit state algorithm is quite efficient on this class of problems. The execution time of the two tools is overall reasonable for many applications [11], [12], though their performance is order of magnitudes higher than the WM approach and prevents their use for run-time verification in highly dynamic environments.

The WM execution time at runtime, i.e. the evaluation of the closed-form expression produced by the design-time partial evaluation discussed before in this section, is shown in Figure 17. Despite the different size of the input models, the runtime of WM is close to constant in these experiments, since the complexity of the parametric expressions mostly depends on the number of parameters rather than the size of the model. Indeed, it is far more efficient in general to evaluate a closed-form expression than solving a system of equations. Finally, notice that also run-time sensitivity analysis described in Section 7 requires the evaluation of a closed-form expression and can thus it be accomplished within similar time complexity.

10.5 Discussion

Hereafter we briefly summarize the main lessons learned both from the validation presented here and from our experience in using WM in practice.

Choosing the right method. A choice has to be made between matrix-based and equation-based methods for the design-time stage of WM. The number of parametric states is the discriminating factor between them. A matrix-based implementation can handle very large systems with a small number of parameters, described by a large transition matrix where only a few rows contain symbolic values. Matrix-based methods split the computation into a symbolic phase and a numeric one. The latter can be then performed through state of the art mathematical engines or routines, able to handle very large matrices (in the order of millions or rows). Both the symbolic and the numeric parts of this approach are straightforwardly parallelizable, thus this approach can be preferred if a parallel execution environment is available. Equation-based methods should be preferred on non parallel environments when the number of parametric states is large, which would make the symbolic computation part of matrix-based approaches too expensive. Finally, the equation-based solver we implemented for WM is optimized for sparse systems, since we expect for DTMC models of a software behavior that each state interacts directly with only a few other states. If the DTMC under analysis is instead dense, other solvers may provide better performance.

Nested formulas. The approaches defined for analyzing nested PCTL formulas cover the full PCTL. However, since they require the introduction of additional parameters, these approaches may be too expensive in terms of memory and execution time as the number of model states grows. The maximum size of the model depends on the available computational resources both at design-time and at runtime. At design-time, a parallel execution of the matrix-based algorithms can reduce the demand for each computation node both in terms of time and memory, though the total number of operations is still exponential in the number of states. At run time, depending on the topology of the model, the expression to evaluate may have a number of terms in the order of $O(n^{\log n})$, where n is the number of states. If the number of states is very large, evaluating such expression may require a non-negligible time, especially on low-power devices (as in the case of Section 8), and a large amount of memory to be stored. This restricts the practical verification of nested formulas to relatively small models.

Numerical issues. The parametric formulae produced by the partial evaluation stage of WM are rational polynomials. The length in number of terms of these polynomials depends on the number of parameters and the topology of the DTMC. To cope with numerical problems, WM uses infinite precision rational numbers for its internal computations. This conservative choice aims at guaranteeing the accuracy and correctness, at the cost of higher time and memory utilization during the design-time partial evaluation. However, translating the parametric formulas into a standard programming language requires additional considerations. First of all, most of the programming languages do not provide native support for rational numbers. This can be added through a variety of libraries, which may lead to increased complexity at run time, especially for low power devices. For our experiments, we generated C code and rational numbers were approximated by floating-point representations. However, for large and complex formulas this translation is

not trivial. The first issue to consider is numerical stability. Indeed, different equivalent forms of the same expression may perform differently in terms of stability [45]. This problem is well known and it is recommended to transform long formulae before implementing them in a specific programming language. Maple provides a code generation function able to optimize the representation of the formula for several programming languages, including C. This optimization consider both numerical stability and time for evaluation, and is the one used by WM. Additionally, most of the compilers can further optimize a mathematical expression for a specific instruction set. For specific applications, ad-hoc representation of the formulas can be designed as a trade-off between accuracy and evaluation time (especially when the computation is carried on low power devices, e.g., embedded systems).

State space reduction. Our WM implementation may further improve its performance by incorporating state space reduction algorithms. Several techniques have been proposed in the field of probabilistic model checking to reduce the size of the model to be analyzed in presence of specific regularities or symmetries (e.g., [86], [84]). Certain techniques perform according to given target properties, while others are general for fragments or the full PCTL. These techniques are used by state of the art model checkers to preprocess the models for simplifying the subsequent analysis. Whenever they can cope with symbolic parameters, these techniques can be used to preprocess the models before applying WM. A notable example is bisimulation reduction, already used by PARAM, which can be implemented for WM too. Bisimulation has been shown to be effective in reducing the number of states of large models having a variety of regularity patterns in their structure [84]; however, in general, it cannot reduce the number of different parameters and preserves the model semantics only with respect to reachability properties. The result of bisimulation is another DTMC having a possibly smaller number of states, but, in general, the same number of parameters.

Modeling software behavior. Without loss of generality (see Section 3.1), in this work we focus on absorbing DTMCs. One might wonder whether and how this choice limits in practice the modeling activity. Absorbing DTMCs have been widely adopted in the area of software to verify probabilistic properties of abstract workflows [25], [36], [87], [56]. As already mentioned, these verification problems can be transformed into equivalent ones on corresponding well-formed DTMCs [19]. They naturally fit the case where a notion of "final" conditions exists in the behavior of the software, for example the completion of the execution or the occurrence of a non-recoverable exception or failure, or a round of a (quasi-)periodic behavior. The model is used to represent an abstract run of the application and the notion of final condition applies to such run. For example, the DTMC in Figure 3 describes the behavior of the Web application for a generic user session. Each of these sessions may either incur in non recoverable errors (e.g., a failed authentication) or terminate.

11 Conclusions and Future Work

In this paper we presented an approach to run-time quantitative verification and sensitivity analysis, developed to support self-adaptive software. The proposed approach relies on the efficient verification of quantitative probabilistic temporal properties on Markov models, exploiting a pre-computation step performed at design time and delaying the evaluation of simple verification conditions that depend on variable parameters to run time. Run-time verification of the residual verification conditions can be performed very efficiently (often in constant time) and can even be performed on low-power devices, such as mobiles devices. In the paper we discussed the mathematical foundations of the proposed solution and we compared the obtained results with the existing competing alternatives, showing a substantial improvement in terms of efficiency.

Our future work will continue on widening the foundations of dependable self-adaptive systems. We will work on extending the techniques discussed here to Markov Decision Processes and Continuous Time Markov Chains. In addition we plan to improve the WM tool and implement the solution algorithms also for open-source mathematical environments. We also plan to improve the tool by adopting ad-hoc pre-processing stages for state space reduction: e.g., bisimulation [84], partial order reduction [88], and SCC decomposition [67]. Finally, the WM approach has been used as enabling feature to bring control theoretical adaptation mechanisms for self-adaptive software with promising

results [89], [90], and we plan to exploit it to enhance automated model learning and controller synthesis techniques for software systems abstracted through DTMC models [91].

ACKNOWLEDGMENT

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

REFERENCES

[1] C. Ghezzi and G. Tamburrelli, "Reasoning on non-functional requirements for integrated services," in *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*. IEEE, 2009, pp. 69–78.

[2] B. H. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 1–26.

[3] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[4] G. Blair, N. Bencomo, and R. B. France, "Models@ run. time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[5] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 111–121.

[6] C. Ghezzi and G. Tamburrelli, "Predicting performance properties for open systems with kami," in *Architectures for Adaptive Software Systems*. Springer, 2009, pp. 70–85.

[7] A. Filieri, L. Grunske, and A. Leva, "Lightweight adaptive filtering for efficient learning and updating of probabilistic models," in *International Conference on Software Engineering*, ser. ICSE. IEEE, 2015.

[8] A. Filieri, C. Ghezzi, and G. Tamburrelli, "A formal approach to adaptive software: continuous assurance of non-functional requirements," *Formal Aspects of Computing*, vol. 24, no. 2, pp. 163–186, 2012.

[9] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli, "Managing non-functional uncertainty via model-driven adaptivity," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 33–42.

[10] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[11] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69—77, Sep. 2012.

[12] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimization in service-based systems," *IEEE Transactions on Software Engineering*, vol. 37, pp. 387—409, 2011.

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

[14] J.-P. Katoen, M. Khattri, and I. S. Zapreev, "A Markov reward model checker," in *QEST*. Los Alamos, CA, USA: IEEE Computer Society, 2005, pp. 243—244.

[15] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 341—350.

[16] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal aspects of computing*, vol. 6, no. 5, pp. 512—535, 1994.

[17] A. Filieri and C. Ghezzi, "Further steps towards efficient runtime verification: Handling probabilistic cost models," in *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, june 2012, pp. 2—8.

[18] S. Andova, H. Hermanns, and J. Katoen, "Discrete-time rewards model-checked," *Formal Modeling and Analysis of Timed Systems*, pp. 88—104, 2004.

[19] C. Baier and J. Katoen, *Principles of model checking*. The MIT Press, 2008.

[20] C. Courcoubetis and M. Yannakakis, "The complexity of probabilistic verification," *J. ACM*, vol. 42, no. 4, pp. 857—907, Jul. 1995.

[21] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *Formal Methods for Performance Evaluation*, ser. Lecture Notes in Computer Science, M. Bernardo and J. Hillston, Eds. Springer, 2007, vol. 4486, pp. 220—270.

[22] W. Farr, "Software reliability modeling survey," *Handbook of software reliability engineering*, pp. 71—117, 1996.

[23] H. Pham, *Software reliability*. Wiley Online Library, 1999.

[24] W.-L. Wang, D. Pan, and M.-H. Chen, "Architecture-based software reliability modeling," *Journal of Systems and Software*, vol. 79, no. 1, pp. 132—146, 2006.

[25] A. Immonen and E. Niemelä, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Software & Systems Modeling*, vol. 7, no. 1, pp. 49–65, 2008.

[26] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, "Large empirical case study of architecture-based software reliability," in *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, nov. 2005, pp. 43—52.

[27] H. Koziolek, B. Schlich, and C. Bilich, "A large-scale industrial case study on architecture-based software reliability analysis," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, nov. 2010, pp. 279—288.

[28] R. C. Cheung, "A user-oriented software reliability model," *Software Engineering, IEEE Transactions on*, no. 2, pp. 118–125, 1980.

[29] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation*, vol. 45, no. 23, pp. 179—204, 2001.

[30] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli, "Mining behavior models from user-intensive web applications," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 277–287.

[31] N. D. Jones, C. K. Gomard, and P. Sestof, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[32] R. Goldblatt, *Logics of time and computation*. Center for the Study of Language and Information, 1995, vol. 60, no. 1.

[33] W. Pestman, *Mathematical Statistics*, ser. De Gruyter Textbook. De Gruyter, 2009.

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

[35] H. Taylor and S. Karlin, *An introduction to stochastic modeling*. Academic Press (Boston), 1994.

[36] M. Kwiatkowska, "Quantitative verification: models techniques and tools," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. ACM, 2007, pp. 449—458.

[37] F. Gantmakher, *The Theory of Matrices*, ser. Chelsea Publishing Series. AMS Chelsea, 2000.

[38] A. Filieri, "Model-based verification and adaptation of software systems @runtime," Ph.D. dissertation, Politecnico di Milano, Milan, Italy, Mar 2013.

[39] 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*. Springer, 1995, pp. 155—165.

[40] L. Grunske, "Specification patterns for probabilistic quality properties," in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, may 2008, pp. 31—40.

[41] S. C. Althoen and R. McLaughlin, "Gauss-Jordan reduction: A brief history," *The American Mathematical Monthly*, vol. 94, no. 2, pp. 130—142, 1987.

[42] A. Bojanczyk, "Complexity of solving linear systems in different models of computation," *SIAM Journal on Numerical Analysis*, vol. 21, no. 3, pp. 591—603, 1984.

[43] F. Rabhi and S. Gorlatch, *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[44] K. Gallivan, M. Heath, E. Ng, J. Ortega, B. Peyton, R. Plemmons, C. Romine, A. Sameh, and R. Voigt, *Parallel Algorithms for Matrix Computations*, ser. Miscellaneous Bks. Society for Industrial and Applied Mathematics, 1987.

[45] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical mathematics*. Springer Verlag, 2007, vol. 37.

[46] D. Parker, "Implementation of symbolic model checking for probabilistic systems," Ph.D. dissertation, University of Birmingham, aug 2002.

[47] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

[48] T. Davis, *Direct methods for sparse linear systems*. Society for Industrial Mathematics, 2006, vol. 2.

[49] V. G. Kulkarni, *Introduction to modeling and analysis of stochastic systems*. Springer, 2011.

[50] G. Golub and C. Van Loan, *Matrix Computations*, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996.

[51] S. Malik and S. Arora, *Mathematical Analysis*. Wiley, 1992.

[52] C. S. Raghavendra, K. M. Sivalingam, and T. Znati, *Wireless sensor networks*. Springer Science & Business Media, 2004.

[53] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-power wireless bus," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*. ACM, 2012, pp. 1–14.

[54] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with glossy," in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE, 2011, pp. 73–84.

[55] M. Zimmerling, F. Ferrari, L. Mottola, and L. Thiele, "On modeling low-power wireless protocols based on synchronous packet transmissions," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*. IEEE, 2013, pp. 546–555.

[56] R. Calinescu and M. Kwiatkowska, "Using quantitative analysis to implement autonomic it systems," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, pp. 100—110.

[57] D. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev, "How fast and fat is your probabilistic model checker? an experimental performance comparison," in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science, K. Yorav, Ed. Springer, 2008, vol. 4899, pp. 69—85.

[58] O. V. Sokolsky and S. A. Smolka, "Incremental model checking in the modal mu-calculus," in *Computer Aided Verification*, 1994, pp. 351–363.

[59] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *Computer Aided Verification*, 2005, pp. 449–461.

[60] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido, "Extreme model checking," in *Verification: Theory and Practice*. Springer, 2003, pp. 332–358.

[61] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 291–300.

[62] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 115–124.

[63] S. Krishnamurthi and K. Fisler, "Foundations of incremental aspect model-checking," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 2, p. 7, 2007.

[64] M. Kwiatkowska, D. Parker, and H. Qu, "Incremental quantitative verification for markov decision processes," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, june 2011, pp. 359—370.

[65] M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "On incremental quantitative verification for probabilistic systems," in *Proceedings of the High-order workshop on automated runtime verification and debugging*, Manchester, UK, 2012.

[66] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "Incremental runtime verification of probabilistic systems," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Qadeer and S. Tasiran, Eds. Springer Berlin Heidelberg, 2013, vol. 7687, pp. 314—319.

[67] F. Ciesinski, C. Baier, M. Grosser, and J. Klein, "Reduction techniques for model checking markov decision processes," in *Quantitative Evaluation of Systems, 2008. QEST '08. Fifth International Conference on*, Sept 2008, pp. 45—54.

[68] I. Meedeniya and L. Grunske, "An efficient method for architecture-based reliability evaluation for evolving systems with changing parameters," in *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, 2010, pp. 229—238.

[69] D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli, "A syntactic-semantic approach to incremental verification," *arXiv preprint arXiv:1304.8034*, 2013.

[70] ——, "Incremental syntactic-semantic reliability analysis of evolving structured workflows," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Springer, 2014, pp. 41–55.

[71] L. Fribourg and A. Étienne, "An inverse method for policy-iteration based algorithms," in *Proceedings International Workshop on Verification of Infinite-State Systems INFINITY*, 2009, pp. 44—61.

[72] E. M. Hahn, T. Han, and L. Zhang, "Synthesis for pctl in parametric markov decision processes," in *Proceedings of the 3rd International Symposium - NASA Formal Methods*, 2011, pp. 146—161.

[73] T. Chen, E. M. Hahn, T. Han, M. Kwiatkowska, H. Qu, and L. Zhang, "Model repair for markov decision processes," in *Proc. 7th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE CS Press, 2013, to appear.

[74] C. Daws, "Symbolic and parametric model checking of discrete-time markov chains," in *Theoretical Aspects of Computing - ICTAC 2004*, ser. Lecture Notes in Computer Science, Z. Liu and K. Araki, Eds. Springer, 2005, vol. 3407, pp. 280—294.

[75] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to automata theory, languages, and computation*. Addison-wesley, 2007.

[76] R. A. Howard, *Dynamic Probabilistic Systems, Volume I: Markov Models*. Courier Dover Publications, 2012.

[77] E. Hahn, H. Hermanns, and L. Zhang, "Probabilistic reachability for parametric markov models," *Model Checking Software*, pp. 88—106, 2009.

[78] ——, "Probabilistic reachability for parametric markov models," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 1, pp. 3—19, 2011.

[79] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang, "Param: A model checker for parametric markov models," in *Computer Aided Verification*, 2010, pp. 660–664.

[80] S. Gokhale and K. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture," in *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, 2002, pp. 64—75.

[81] V. Cortellessa and V. Grassi, "A modeling approach to analyze the impact of error propagation on reliability of component-based systems," *Component-Based Software Engineering*, pp. 140—156, 2007.

[82] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," *Component-Based Software Engineering*, pp. 1—20, 2010.

[83] G. Su and D. S. Rosenblum, "Perturbation analysis of stochastic systems with empirical distribution parameters," in *International Conference on Software Engineering*, ser. ICSE. ACM, 2014, pp. 311–321.

[84] J.-P. Katoen, T. Kemna, I. Zapreev, and D. N. Jansen, "Bisimulation minimisation mostly speeds up probabilistic model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 87–101.

[85] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Springer Berlin Heidelberg, 2011, vol. 6806, pp. 585—591.

[86] ——, "Symmetry reduction for probabilistic model checking," in *Computer Aided Verification*, 2006, pp. 234–248.

[87] C. Ghezzi and A. M. Sharifloo, "Model-based verification of quantitative non-functional properties for software product lines," *Information and Software Technology*, 2012.

[88] C. Baier, P. D'Argenio, and M. Groesser, "Partial order reduction for probabilistic branching time," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 97–116, 2006.

[89] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, "Reliability-driven dynamic binding via feedback control," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*. IEEE, 2012, pp. 43–52.

[90] ——, "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 283–292.

[91] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 299–310.