Under consideration for publication in Formal Aspects of Computing

A Formal Approach to Adaptive Software: Continuous Assurance of Non-Functional Requirements

Antonio Filieri and Carlo Ghezzi and Giordano Tamburrelli
DEI—Politecnico di Milano,
DeepSE Group,
Piazza L. da Vinci, 32
20133 Milano, Italy

Abstract. Modern software systems are increasingly requested to be adaptive to changes in the environment in which they are embedded. Moreover, adaptation often needs to be performed automatically, through self-managed reactions enacted by the application at run time. Off-line, human-driven changes should be requested only if self-adaptation cannot be achieved successfully. To support this kind of autonomic behavior, software systems must be empowered by a rich run-time support that can monitor the relevant phenomena of the surrounding environment to detect changes, analyze the data collected to understand the possible consequences of changes, reason about the ability of the application to continue to provide the required service, and finally react if an adaptation is needed. This paper focuses on non-functional requirements, which constitute an essential component of the quality that modern software systems need to exhibit. Although the proposed approach is quite general, it is mainly exemplified in the paper in the context of service-oriented systems, where the quality of service (QoS) is regulated by contractual obligations between the application provider and its clients. We analyze the case where an application, exported as a service, is built as a composition of other services. Non-functional requirements—such as reliability and performance—heavily depend on the environment in which the application is embedded. Thus changes in the environment may ultimately adversely affect QoS satisfaction. We illustrate an approach and support tools that enable a holistic view of the design and run-time management of adaptive software systems. The approach is based on formal (probabilistic) models that are used at design time to reason about dependability of the application in quantitative terms. Models continue to exist at run time to enable continuous verification and detection of changes that require adaptation.

Keywords: Software evolution, (self)adaptive software, non-functional requirements, reliability, performance, models, model-driven development, Markov models, verification, probabilistic model checking, monitoring, Bayesian inference.

Correspondence and offprint requests to: Antonio Filieri, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza L. da Vinci, 32, 20133 Milano, Italy. e-mail: filieri@elet.polimi.it

1. Introduction

Modern software applications are increasingly embedded in an *open world* [6], characterized by *continuous change* in the environment in which they are situated and in the requirements they have to meet. Changes are not under the application's control; they occur autonomously and unpredictably. At the same time, there is a growing demand for software solutions that can recognize and tolerate changes, by evolving and dynamically adapting their behavior to preserve quality of service as changes occur. This is especially needed when systems must be perpetually running and cannot be taken off-line to be updated. If the software is unable to adapt autonomously, it must at least provide high-level feedback information to a software engineer in the loop to support off-line adaptation [18].

This paper focuses on *environment changes* that may affect the quality of an application. For example, in an embedded systems domain, the arrival rate of certain control data may change because of an unanticipated physical phenomenon. Likewise, the error rate of data streamed by sensing devices may change over time because of battery consumption. In a business applications domain, changes may instead concern the way people interact with the system or the integrated external components, which offer services upon which the currently developed application relies. The latter changes may become visible when new versions of such components are deployed and made available for use to the rest of a system.

This paper also focuses on non-functional requirements [62], and in particular on *performance* and *reliability*, which heavily depend on external factors that occur in the environment, like in the previous examples. These factors are hard to understand and predict when the application is initially designed. Moreover, even if the predictions are initially accurate, they are likely to change continuously as the application is running. As an example, consider the number of requests per second of a given purchase transaction submitted by the users of an e-commerce application. At design time, the software engineer can only predict these values based on her past experience or the observed behavior of similar systems. But these may not exist, or they may differ substantially from the application being designed. Furthermore, sudden changes in user profiles during certain periods of the year (e.g., Christmas or national holidays) may invalidate the assumptions upon which the system was initially designed. These assumptions, unfortunately, may be crucial to assuring the desired quality. For example, the number of requests per second that can be handled by the e-commerce system may affect its performance (the average response time for user requests) and even its reliability (the requests that cannot be handled are dropped and this may generate failures on the client's side). As another example, consider a mobile pervasive computing application, where the network latency may fall below a certain threshold when the user moves to certain indoor locations, because of physical obstacles between stations.

It is ever more important that non-functional requirements be stated in a precise and possibly formal manner, so that we can manage *quality of service (QoS)* in a precise and mathematically well-founded way. A precise definition of QoS should indeed characterize the *dependability contract* between an application and its clients. Dependability is defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as:

the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers[1]

For example, in *service-oriented computing (SOC)*, where applications are built by composing components (services) that are owned, deployed, and run by external entities, a *service-level agreement (SLA)* defining the dependability contract of each external provider must be stated in order for the application to also be able to define its own dependability contract with its clients. Likewise, the requirements satisfied by an embedded system are formalized in a specification document that defines both the required behavior (functional requirements) and non-functional aspects, such as reliability or response-time bounds.

To express non-functional requirements formally, one needs to take into account the *uncertainty* that characterizes them, as discussed earlier. Uncertainty may be formally and quantitatively stated in probabilistic terms. For example, focusing on performance, a given probability distribution may be provided to specify the response time of a certain external service or input data source. Likewise, reliability of an external component can be stated in terms of its expected failure rate. For this reason, in this work we focus on probabilistic Markov models of the application, namely *Discrete-Time Markov Chains (DTMCs)* and *Continuous-Time Markov Chains (CTMCs)*, briefly summarized in Section 3 .

According to the traditional software engineering principles, modeling is considered as an important

[1] http://www.dependability.org/

conceptual tool in the requirements and design stages of software development. Requirements models are built to better understand and reason about the qualities a system should exhibit in order to fulfill its goals. They are also used to support systematic interaction with the stakeholders in requirements elicitation. Modeling also helps crystallize design decisions and evaluate the trade-offs among them. Model-driven development has also been proposed [10], and several approaches are being investigated, with the goal of automating the generation of implementations through chains of (possibly automated) model transformation steps [20].

We argue, however, that the role of models is even more important in the context of adaptive software. In fact, we concur with other authors [11] that models should be kept alive at run-time to support *continuous verification*. Continuous verification is needed because:

- the incomplete and uncertain knowledge about the environment available at design-time may be invalidated by the real-world behaviors observed later, when the application becomes operational and provides service;
- even assuming that design-time knowledge is initially accurate, it may evolve later, and subsequent evolution may invalidate the assumptions upon which it stands.

Continuous verification is responsible for on-line assessment that the running system behaves as stated in the requirements specification. It may do so by:

- *Monitoring* the behavior of the external world, by collecting run-time data from the external entities whose behavior may change and may affect the the overall application's behavior.
- Understanding whether the data collected by the monitor point to changes in the models originally used for verification. This phase may be supported by suitable *machine learning* approaches.
- *Verifying* that the updated models continue to meet the requirements.

Continuous verification provides the formal foundations for adaptation. By detecting a violation of the requirements of the updated model, the need for an adaptation is automatically identified. If possible, the system should try to self-adapt to changes. If self-adaptation is not possible, the software engineer may inspect the formal model and get insights into how to perform off-line adaptation.

The purpose of this paper is to provide a holistic approach to lifelong verification of reliability and performance properties of evolving and adaptive software. We do this by integrating and extending some of our previous work [22, 30, 29]. The integrated approach, called KAMI (*Keeping Alive Models with Implementation*), is supported by a prototype that we developed.

The paper is structured as follows. Section 2 provides a reference framework, which allows us to precisely talk about changes and adaptation. Section 3 recalls the necessary background about the formal models (DTMCs and CTMCs) and the property languages (PCTL and CSL) we use in the paper. Section 4 illustrates a running example that will be used subsequently to present our approach. We first discuss how the initial models for the running example may be verified at design time to assess reliability and performance properties. In Section 5 we then introduce our approach to continuous verification and we show how run-time verification can be achieved for the running example. Section 6 shows our approach in action, also through simulations. Section 7 describes related work. Finally, Section 8 draws some conclusions and outlines future work directions.

2. A Reference Framework for Change and Adaptation

The software developed by engineers provides an abstract machine whose goal is to fulfill certain requirements in the world in which it is embedded. M. Jackson and P. Zave, in their seminal work on requirements [39, 65], clearly distinguish between the *world* and the *machine*. The machine is the system to be developed; the world (the environment) is the portion of the real-world affected by the machine. The ultimate purpose of the machine—the *requirements*—are always to be found in the world. Thus they are expressed in terms of the phenomena occurring in the world, as opposed to phenomena occurring inside the machine. Some of the world phenomena are shared with the machine: they are either controlled by the world and observed by the machine, or controlled by the machine and observed by the world. A *specification* (for the machine) is a set of prescriptive statements on the relation on shared phenomena that must be enforced by the system to be developed. To achieve requirements satisfaction, the machine relies on assumptions on the environment, captured as *domain knowledge*[2]. Quoting from [65], "The primary role of domain knowledge is to bridge the gap between requirements and specifications."

Domain knowledge captures the set of relevant assumptions that need be made about the environment in which the machine is expected to work in order to be able to prove that—through the machine—we achieve the desired requirements. Let R and S be prescriptive statements in some formal notation that describe the requirements and the specification, respectively, and let D be the descriptive formal statements that specify the domain assumptions. If S and D are all satisfied and consistent, then it should be possible to prove that R also holds:

$$S, D \models R \tag{1}$$

That is, S ensures satisfaction of the requirements R in the context of the domain properties D.

Although the framework presented here and the methodology illustrated in the rest of the paper are quite general and can be applied to a wide range of domains—from embedded systems to business applications—we especially focus here on *user-intensive* interactive applications. This means that the behavior of users in submitting their requests to the software—such as the distribution profile of different requests and the submission rates—affects QoS. In addition, it is a primary source of potential change of the application's non-functional qualities. Other domain-specific environment phenomena affect quality of other kinds of systems (e.g., embedded or pervasive systems). We further assume that the application integrates *external services* provided by independent service providers; that is, we position our work in a typical *service-oriented computing (SOC)* setting, which is becoming quite common and therefore has a great practical relevance. Externally available services provided by third parties are viewed as *black boxes*: they can be invoked by the application, but their internals are unknown. Their behavior and QoS are only known through provider-supplied interfaces. A well-known and practically relevant example is provided by the Web services world, where external services are composed through a service composition (or workflow) language, such as BPEL [2, 12]. A typical situation is illustrated in Figure 1(a), which provides a high-level view of a workflow integrating external services.

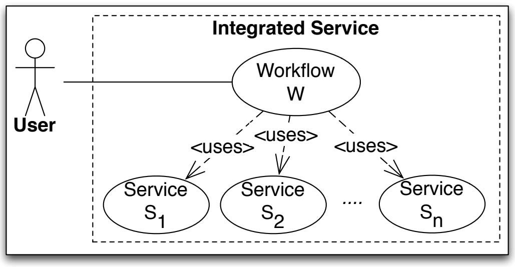
Let us investigate how Equation 1 may be further detailed in this setting. We can partition the relevant set of domain properties D into three disjoint subsets, D_f, D_u, and D_s. D_f is the fixed part: it captures the set of stable assumptions, which will not change later. For example, D_f may include the known physical laws that regulate certain environment phenomena. D_u and D_s instead capture the assumptions that are likely to change over time. At run time, the changeable parts of the environment must be monitored to check if the assumptions D_u and D_s are violated and if their violation yields to a violation of R.

D_u denotes the *assumptions on usage profiles.* They consist of properties that characterize how the application being designed is expected to be used by its clients. D_u properties must be taken into account during design because they may drive architectural decisions that lead to satisfaction of non-functional requirements. However, the assumptions made at design time may turn out to diverge from what the running system eventually exhibits. Furthermore, they may change dynamically. Going back to the e-commerce example introduced earlier, the usage profile may concern the average number of purchase transactions per registered user per day. Besides being hard to predict, this value is also subject to high variability. For example, it may be discovered during operation that the volume of purchases during the two weeks before Christmas is 5 times the average volume of all other weeks of the year, and unfortunately this leads to transactional volumes that are incompatible with the initial architectural design.

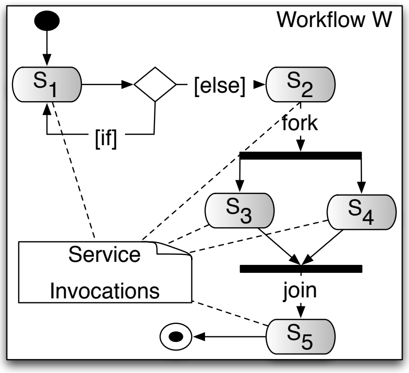
D_s instead denotes the set of *assumptions on the external services* invoked by the application. If $D_{s,1}, D_{s,2}, \ldots, D_{s,n}$ are the assumptions on the required behavior to be provided by individual external services, $D_s = D_{s,1} \wedge D_{s,2}, \ldots \wedge D_{s,n}$. These assumptions must of course be matched by the corresponding *specifications* $S_1, S_2, \ldots S_n$ of the external services to be composed in the application's implementation. These specifications may be viewed as the dependability contracts that constrain service providers through the SLA.

We do not make any specific assumptions on the languages in which one can express the statements R, D, S. No commonly used notations, let alone standards, exist for them. For example, in the domain of Web services, the WSDL [64] specification language only focuses on syntactic aspects of a service interface, and hence it does not fit our purpose. Attempts to support specification of quality attributes, such as [60],

[2] We use the terms *requirements*, *specification*, and *domain knowledge* as in Jackson and Zave's work. Other authors (e.g., [62]) follow different conventions.



(a) Integrated services scenario.



(b) Workflow Example

Fig. 1. Integrated Services

are still not adopted in practice. Contract languages [54] were proposed to specify module interfaces; for example, JML [13], Spec# [7], and Microsoft Contracts [24]. These languages are semantically richer and formal, but they lack support for non-functional properties.

To support reasoning about requirements satisfaction (both at design time and at run time), we rather take a more pragmatic approach that does not require the definition of a new ad-hoc notation for R, S, and D, but instead chooses notations that can be supported by existing analysis tools. Our approach is centered around the use of the PRISM [36, 47] *stochastic model checker*, but it can be easily adapted to any other model-checker supporting the used model and logic formalisms. PRISM can check properties (corresponding to R) written in probabilistic temporal logics. The models against which such properties are checked are *state-transition systems*, which provide an operationalized description of S. The operational models are augmented with probabilities, yielding stochastic processes represented as *Markov models*. Stochastic annotations on transitions are used to express the domain assumptions corresponding to D_s and D_u. We will use two different Markov models to reason about reliability and performance, respectively: *Discrete-Time Markov Chains* and *Continuous-Time Markov Chains*. We also use different *probabilistic temporal logic languages* to express reliability-related and performance-related properties that capture R: Probabilistic Computation Tree Logic (*PCTL*) and Continuous Stochastic Logic (*CSL*) [5]. Markov models and probabilistic logics are briefly recalled in the next section.

In this paper, we start at the point in which a model—describing both the behavior of the application (S) and of the environment (D)—is given and we wish to analyze whether it represents a requirements compliant system. We do not address the important problem of how and when the model may be derived and how and why one decides which is the its abstraction level. This is outside the scope of this paper, and is actually one of our group's current research directions, which focuses on the requirements engineering process for adaptive systems. However, our choice to refer to Markov models should not surprise the reader, since they are close to notations familiar to practitioners. In a nutshell, Markov models are simply conventional state-transition systems with annotations on transitions through which additional non-functional aspects can be specified. State-transition systems are commonly used in practice by software designers and can be used at different levels of abstraction to model and analyze software systems [57]. Alternatively, the initial model can be expressed as an activity diagram or any other control-flow model of the application. Several approaches are available to support automatic model-to-model transformation from activity diagrams to Markov models (see, for example, [19, 26]) In Section 4 we will provide an example of a specification given in terms of an activity diagram, which is transformed into a DTMC modeling both S and D from a reliability standpoint, and we will show how the DTMC can be used to assess the entailment of R.

Let us discuss now what happens at run time if a violation of S or D is detected. A violation of S is a consequence of an implementation flaw. A violation of D is instead due to an environmental change, i.e., a change of either D_u or D_s or both. In this paper, we deliberately ignore implementation flaws and focus instead on environment changes. Three possible situations may arise when D is violated:

1. Environment changes that lead to changes of D do not imply a violation of the requirements; that is,

statement (1) can still be proved. If the violation is due to a change of some $D_{s,i}$, it means that we detect a *failure* of service i: its provider breaks the SLA, and may thus undergo a penalty. However, in this case the implementation is robust enough to tolerate the failure of the external service, which has no consequence on satisfaction of R.

2. Environment changes that lead to changes of D also lead to a violation of R. Here we can distinguish between two possible sub-cases:
   (a) The violation of requirements R is detected by the model checker, but the violation is not experienced in the real world. This is the case of a *failure prediction* by the model checker. For example, a detected change in the response time of an external service may cause a violation of a required performance property of a certain operation which is detected by the model checker even though the operation has not been invoked yet by users in the real world. The model checker predicts that a failure will be experienced when the operation will be invoked.
   (b) The violation of R detected by the model checker is also experienced in the external world. In this case the model checker performs a *failure detection.*

Another way to interpret this situation is through the concepts of *fault* and *failure* used by the dependability community [3]. A failure is defined as a manifestation of a fault. A violation of R detected by the model checker at run time indicates that a fault has occurred; it does not necessarily indicate a failure. By identifying the fault before it leads to a failure, we have an opportunity to react by preventing its manifestation. Finally, we may say that any adaptive reaction that tries to modify S—the machine—as a consequence of a violation of R is a *preventive action* in the case of failure prediction, and a *repairing action* in the case of failure detection.

3. Background Mathematical Foundations

In this section we provide a brief introduction to the mathematical concepts used throughout the paper: DTMCs, CTMCs, and probabilistic logics. The reader can refer to [4, 5] for a comprehensive in-depth treatment of these concepts and for a presentation of the corresponding model checking algorithms.

3.1. Discrete Time Markov Chains

Discrete Time Markov Chains (DTMCs) are a widely accepted formalism to model reliability of systems built by integrating different components (services). In particular, they proved to be useful for an early assessment or prediction of reliability [38]. The adoption of DTMCs implies that the modeled system meets, with some tolerable approximation, the Markov property–described below. This issue can be easily verified as discussed in [19, 31].

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

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

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

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

3.2. Probabilistic Computation Tree Logic and reliability properties

Formal languages to express properties of systems modeled through DTMCs have been studied in the past and several proposals are supported by model checkers to prove that a model satisfies a given property. In this paper, we focus on Probabilistic Computation Tree Logic (PCTL) [5], a logic that can be used to express a number of interesting reliability properties.

PCTL is defined by the following syntax:

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

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

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

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

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

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

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

$$\diamond^{\leq t}\phi \ \equiv \ true \ U^{\leq t}\phi$$

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

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

3.3. Continuous Time Markov Chains

Continuous Time Markov Chains (CTMCs) are stochastic models that allow one to express the execution time of each step of the process. Each execution time is characterized by means of an exponential distribution. Formally, a CTMC [40, 4] is a tuple $(S, S_0, \mathbf{R}, L)$ where:

- S, S_0, and L are defined as for DTMCs.
- $\mathbf{R} : S \times S \to \mathbb{R}_{\geq 0}$ is the *rate matrix*. An element $\mathbf{R}(s_i, s_j)$ represents the rate at which the process moves from state s_i to state s_j.

An *absorbing state* s_i is a state such that $\mathbf{R}(s_i, s_j) = 0$ for all states s_j. For any pair of states s_i, s_j $\mathbf{R}(s_i, s_j) > 0$ iff there exists a transition from s_i to s_j. Furthermore, $1 - e^{-\mathbf{R}(s_i,s_j)\cdot t}$ is the probability that the transition from state s_i to state s_j is taken within t time units. Through rates, we model the delay in choosing the transition. If more than one transition exiting state s_i has an associated positive rate, a *race* among transitions occurs. The probability $\mathbf{P}(s_i, s_j)$ that control, held by s_i, will be transferred to state s_j corresponds to the probability that the delay of going from s_i to s_j is smaller than the one of going toward any other state; formally, $\mathbf{P}(s_i, s_j) = \mathbf{R}(s_i, s_j)/E(s_i)$, where $E(s_i) = \sum_{s_j \in S} \mathbf{R}(s_i, s_j)$. $E(s_i)$ is called the *exit rate* of state s_i.

This leads to the notion of an *embedded* DTMC for a CTMC. For a CTMC $\mathcal{C} = (S, S_0, \mathbf{R}, L)$, the embedded DTMC $\mathcal{D}$ is defined by $(S, S_0, \mathbf{P}, L)$, where $\mathbf{P}(s_i, s_j) = \mathbf{R}(s_i, s_j)/E(s_i)$ if $E(s_i) > 0$. In case $E(s_i) = 0$, $\mathbf{P}(s_i, s_i) = 1$ and $\mathbf{P}(s_i, s_j) = 0$ for all $s_j \neq s_i$.

We will refer to the embedded DTMC in Section 5, where we will show that estimating the parameters of a CTMC is reduced to estimating the transition probabilities of the underlying DTMC and the exit rate of each state.

3.4. Continuous Stochastic Logic and performance properties

Since CTMCs are defined over a continuous time domain, it is convenient to define a path by non only describing the sequence of states, but also the "delay" between one step and the following one. A common representation for a path π is therefore a sequence $s_0 t_0 s_1 t_1 s_2 ...$ [40], meaning that π starts in state s_0 and spends there t_0 time units, then moves to s_1, and so on. Notice that each t_i is a strictly positive real value, according to the previous definition of a CTMC.

In this paper we consider Continuous Stochastic Logic [4] to state properties on CTMC models. For CTMC there are two main type of properties relevant for analysis: *steady-state* properties where the system is considered "on the long run", that is, when an equilibrium has been reached; *transient* properties where the system is considered at specific time points or intervals. CSL is able to express both steady-state and transient properties by means of the $\mathcal{S}$ and $\mathcal{P}$ operators, respectively. The syntax of CSL is recursively defined as follows:

$$\Phi ::= true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{S}_{\bowtie p}(\Phi) \mid \mathcal{P}_{\bowtie}(\Psi)$$

$$\Psi ::= X^{\leq t}\Phi \mid \Phi U^{\leq t}\Phi$$

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

The semantics of boolean operators is the same defined for PTCL, as well as the possibility to define derived operators, both boolean and temporal. To address the semantics of $\mathcal{S}$ and $\mathcal{P}$, let us first introduce the notation $\pi@t$, which denotes the state in which the execution described by trace π is at time t. As before, $\pi[i]$ denotes the i-th state the execution passed through, thus $\pi@t = \pi[i]$ with $i = \min_i\{t \leq \sum_{j=0}^{i} t_j\}$.
In analogy with PCTL, path formulae's satisfaction is defined as follows:

$$\pi \models X^{\leq t}\Phi \;\text{ iff }\; \pi[1] \text{ is defined and } \pi[1] \models \Phi \text{ and } \delta(\pi, 0) \leq t$$

$$\pi \models \Phi_1 U^{\leq t}\Phi_2 \;\text{ iff }\; \exists\, 0 \leq x \leq t\ (\pi@x \models \Phi_2 \wedge\ \forall\, 0 \leq y < x\ \pi@y \models \Phi_1)$$

where $\delta(\pi, i)$ is the time spent in the i-th state appearing in path π. The formal definition of satisfaction for $\mathcal{S}$ or $\mathcal{P}$ would force us to go through some mathematical intricacies that can be avoided here for sake of readability. The interested reader can refer to [4] to find all the details. Hereafter we simply provide an informal and intuitive description. As for PCTL, the informal semantics of a $\mathcal{P}_{\bowtie p}(\Psi)$ formula is given by the following assertion: the probability that the set of paths starting in s and satisfying Ψ meets the bound $\bowtie p$. As for a steady-state formula $\mathcal{S}_{\bowtie p}(\Phi)$, its informal semantics is expressed by the following assertion: the probability of being in a state where Φ holds when $t \to \infty$, evaluated on all paths originating in s, meets the bound $\bowtie p$.

CSL can naturally represent performance-related properties for a CTMC model of the system, such as its latency or throughput. For example, we can express the property that the probability of completing a transaction (i.e., reaching a given final state from an initial state) within a given time limit is greater than a certain value. More examples will be given next.

4. A Running Example, Initial Model and Verification

In this section we introduce an example, which is used throughout the remaining sections. We also show how the example application is initially developed, starting from an identification of its requirements and of the uncertain and/or changeable domain properties. Following the discussion in Section 2, we show how the high-level models of the application, which embody both S (the high-level software specification) and D (the assumptions on the environment phenomena that affect the system under development), allow us to reason about satisfaction of reliability and performance requirements at design time. Run-time management of non-functional requirements is discussed later in Section 5.

The example, adapted from [30], is described by the activity diagram of Figure 2. The diagram describes a workflow representing a typical e-commerce application that sells on-line merchandise to its customers by integrating the following third-parties services: (1) *Authentication Service*, (2) *Payment Service*, and (3) *Shipping Service*. Invocation of third-party services is graphically indicated by shaded boxes in the activity diagram. Figure 2 is an abstract description of the integrated service, given by using a technology-neutral notation (as opposed, for example, to a specific workflow language like BPEL [2, 12]). The *Authentication Service* manages the identity of users[3]. It provides a *Login* and a *Logout* operation through which the system authenticates users. The *Payment Service* provides a safe transactional payment service through which users can pay for the purchased items. A *CheckOut* operation is provided to perform such payments. The *Shipping Service* is in charge of shipping goods to the customer's address. It provides two different operations: *NrmShipping* and *ExpShipping*. The former is a standard shipping functionality while the latter represents a faster and more expensive alternative. The system also classifies the logged users as *ReturningCustomers* (RC) or *NewCustomers* (NC), in order to provide targeted advertisements and personalized interfaces. The classification determines the binding of the same buying operations to different designs and allows for a finer profiling of customers' behavior by partitioning the user set.

Let us consider the overall non-functional requirements that must be satisfied by the e-commerce transaction. Examples of reliability requirements are:

- *R1: "Probability of success shall be greater than 0.8"*
- *R2: "Probability of a ExpShipping failure for a user recognized as a returning customer shall be less than 0.035"*
- *R3: "Probability of an authentication failure shall be less than 0.06"*

Examples of performance requirements are:

- *P1: "At least 95% of the sessions (from login to logout) shall require at most 1 second of computation time."*
- *P2: "No more than 5% of the sessions of a returning customers shall take more than 0.55 seconds of computation time."*
- *P3: "A new customer shall be able to complete her session up to logout, through normal shipping, waiting no more than 1 second, in no less than the 65% of the sessions."*

Concerning the domain knowledge D_u, Table 1 summarizes what domain experts or previous similar systems tell us. The notation $Pr(x)$ denotes the probability of "x". Table 2 describes the properties that characterize the domain knowledge about external services (i.e., D_s). $PF(Op)$ here denotes the probability of failure of

[3] an example of this functionality is provided by [56].

Table 1. Domain Knowledge D_u: usage profile.

$D_{u,n}$	Description	Value
$D_{u,1}$	*Pr(User is a RC)*	0.35
$D_{u,2}$	*Pr(RC chooses express shipping)*	0.5
$D_{u,3}$	*Pr(NC chooses express shipping)*	0.25
$D_{u,4}$	*Pr(RC searches again after a buy operation)*	0.2
$D_{u,5}$	*Pr(NC searches again after a buy operation)*	0.15

Table 2. Domain Knowledge D_s: failure probabilities (FP).

$D_{s,n}$	Description	Value
$D_{s,1}$	*FP(Login)*	0.03
$D_{s,2}$	*FP(Logout)*	0.03
$D_{s,3}$	*FP(NrmShipping)*	0.05
$D_{s,4}$	*FP(ExpShipping)*	0.05
$D_{s,5}$	*FP(CheckOut)*	0.1

service operation Op. Table 3 provides the expected exit rate for each service, which represents, as explained in Section 3.3 the average number of requests processed in a time unit.

As we recalled, the software engineer's goal is to derive a specification S that leads to satisfaction of requirements R, assuming that the environment behaves as described by D. Starting from the activity diagram of Figure 2, taking into account the stated user profiles and the domain knowledge expressed in Tables 1, 2, 3, we can easily derive analyzable Markov models. That is, by merging the behavioral specification (S) defined by the activity diagram with our domain knowledge (D), we can derive target models that can then be analyzed to check the properties of interest. In order to reason about reliability properties, the analytical model coming out of S and D is a Discrete Time Markov Chain, illustrated in Figure 3. The DTMC contains one state for every operation performed by the system plus a set of auxiliary states representing potential failures associated with external service invocations (e.g., state 5) or specific internal system sates (e.g., state 2). Notice that the failure states cannot be left once accessed, i.e. they are *absorbing.* Figure 4 illustrates the CTMC that represents the application's performance model, which merges the machine's specifications and the performance domain assumptions listed in Table 3. As mentioned, several approaches are available to support automatic model-to-model transformations, including those from activity diagrams to Markov models. An example is the ATOP tool [26].

By using the models in Figures 3 and 4 it is possible to verify satisfaction of the requirements. To accomplish this task we use automatic model checking techniques.

Table 3. Domain Knowledge D_p: exit rates (λ).

$D_{p,n}$	Description	Value (reqs/sec)
$D_{p,1}$	λ*(Login)*	15
$D_{p,2}$	λ*(Profiler)*	15
$D_{p,3}$	λ*(ReturningCustomersInterface)*	20
$D_{p,4}$	λ*(NewCustomerInterface)*	10
$D_{p,5}$	λ*(ReturningCustomerSearch)*	30
$D_{p,6}$	λ*(NewCustomerSearch)*	10
$D_{p,7}$	λ*(ReturningCustomerBuy)*	15
$D_{p,8}$	λ*(NewCustomerBuy)*	10
$D_{p,9}$	λ*(ExpressShipping)*	20
$D_{p,10}$	λ*(NormalShipping)*	25
$D_{p,11}$	λ*(Checkout)*	20
$D_{p,12}$	λ*(Logout)*	15

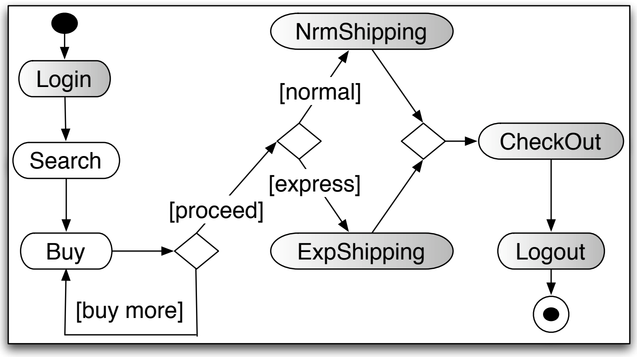


Fig. 2. Operational Description of the Specification via an Activity Diagram

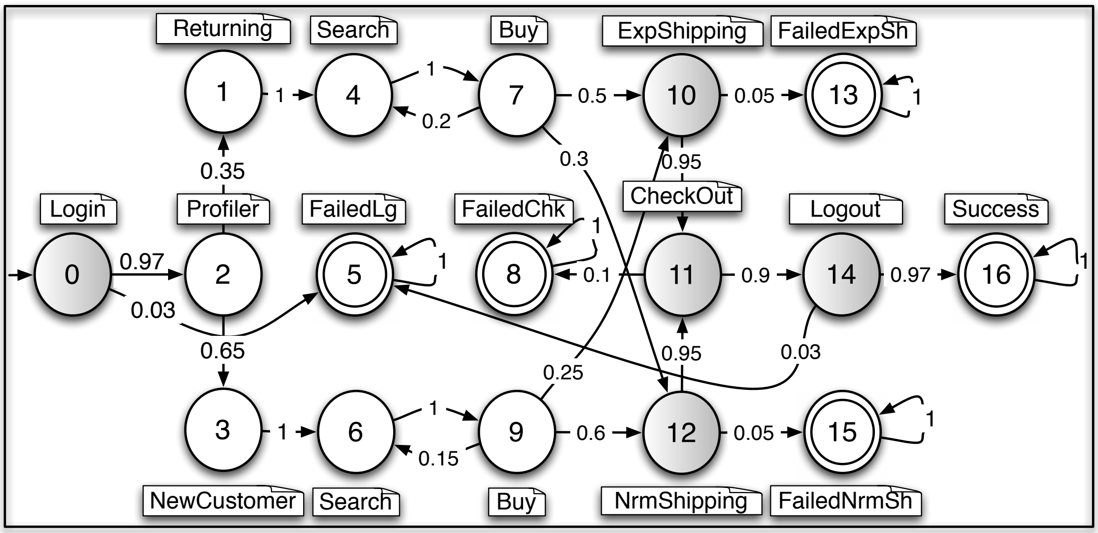


Fig. 3. Example DTMC Model.

Reliability requirements R1-R3 can be translated in PCTL as follows (please notice that all properties are assumed to be evaluated in the initial state, unless a starting state is explicitly stated within curly brackets)[4]:

- $\mathcal{P}_{>0.8}$ [$\Diamond$ s = 16]: the probability of eventually reaching state 16, which corresponds to the successful completion of the session (see Figure 3) is greater than 0.8.
- $\mathcal{P}_{<0.035}$ [$\Diamond$ $s = 13$ $\{s = 1\}$]: the probability of eventually reaching state 13 (representing a failed express shipping) given that the DTMC starts its execution in state 1 (*Returning Customer*) is less than 0.035.
- $\mathcal{P}_{<0.06}$ [$\Diamond$ s = 5]: the probability of eventually reaching the *FailedLg* state is less than 0.06.

By using the probabilistic model checker PRISM [36, 47] on the example to verify the satisfaction of the previous properties, we get the following probabilities:

- *"Probability of success is equal to 0.804"*
- *"Probability of a ExpShipping failure for a user recognized as returning customer is equal to 0.031"*

[4] Instead of characterizing states by atomic propositions, for readability reasons, we use state labels of the form $s = K$ meaning that the process is in state K.

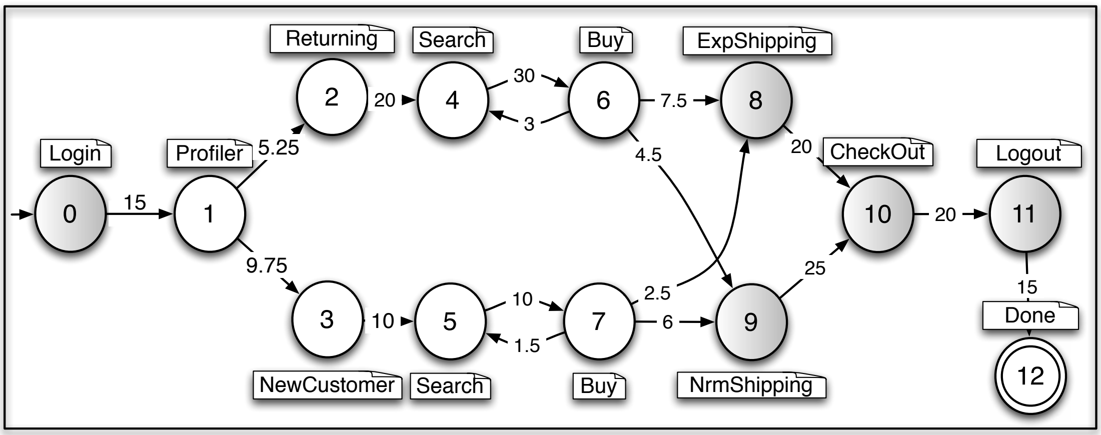


Fig. 4. Example CTMC Model (labeled by exit-rates).

- *"Probability of an authentication failure is equal to 0.055"*

These results prove that reliability requirements are satisfied by the model.

Performance requirements have been mapped into the following CSL formulae (notice that the notion of time here is continuous and not discretized in time steps as for PCTL):

- $\mathcal{P}_{>0.95}$ $[\Diamond^{\leq 1}\ s = 11]$: the probability to reach state 11 (*Logout*) within 1 second is greater than 0.95.
- $\mathcal{P}_{>0.95}$ $[\Diamond^{\leq 0.55}\ s = 11\ \{s = 2\}]$: the probability for a returning user (state 2 identifies a user as RC) to wait less than 0.55 seconds is greater than 0.95. This property is much easier to be expressed in CSL than the human-readable formulation of Section 4, though they are of course equivalent.
- $\mathcal{P}_{\geq 0.65}$ $[((s \neq 8)\ U^{\leq 1}\ s = 11)\ \{s = 3\}]$: literally, a user recognized as NC (state 3 identifies a user as NC) has to reach the *Logout* state within 1 second, without passing through *ExpShipping* with probability higher than 0.65. By looking at Figure 4 it is clear that this corresponds to considering only sessions that include a *NrmShipping*.

Again, by means of PRISM, we get the following probabilities:

- *"Probability that a session (from login to logout) requires at most 1 sec of computation time is equal to 0.966."*
- *"Probability that the computation time needed for a returning customer up to checkout is less than 0.55 seconds is equal to 0.963"*
- *"Probability that a new customer completes her session up to logout through NormalShipping waiting no more than 1 second is equal to 0.692"*

We can therefore conclude that our design satisfies the requirements under the stated assumptions on the behavior of the environment.

5. Run-Time Support to Verification and Adaptation

This section focuses on run-time support to verification and adaptation. As we discussed previously, at design time we need to make assumptions on the external environment in order to be able to prove that the specified machine satisfies its performance and reliability requirements. At run time, after the application is deployed and running, the implemented machine (assumed to be correct with respect to its specification) interacts with the real world, which may behave differently from the initial assumptions. The possibly different behaviors must be detected and analyzed; if necessary, a suitable adaptation must be activated if necessary to keep satisfying the requirements. More precisely, the following main run-time actions must be performed:

1. Environment data must be collected through *monitoring.* From D_u and D_s we can derive which data exchanged with the environment must be collected, because they affect the domain properties upon which we rely to establish satisfaction of the requirements.
2. From the collected data we must be able to infer which model parameters have changed, if any. This is a machine learning step through which we do *model retrofitting.*
3. The updated model, which is kept alive at run time, is re-run to perform *continuous verification.* Verification may succeed—i.e., changes do not violate the requirements—or it may fail.
4. A verification failure may be interpreted either as a *failure detection* or as a *failure prediction*, as we discussed in Section 2. A suitable adaptive self-reaction may be triggered accordingly.

A discussion of monitoring and adaptive self-reactions falls outside the scope of this paper. As for adaptation, in this paper we restrict our focus to identifying the changes in the environment that require adaptation. Adaptation strategies—a currently active research area—are briefly discussed in the related and future work section. In the sequel we do an in-depth analysis of the run-time steps that concern model retrofitting and run-time verification.

5.1. Model Retrofitting with Reliability Parameters

Hereafter we discuss how a DTMC can be automatically updated by observations of run-time data collected from the environment. We assume that the given DTMC models a system that interacts with an external environment in which certain phenomena that can affect the application's reliability may change over time. The data concerning the relevant environment phenomena, such as the detected failures of external services, are collected and used to infer an updated DTMC, which reflects the changes detected in the environment. The method is based on a Bayesian inference technique, and was originally proposed in [22].

Let $\mathbf{P}$ be the transition matrix of the DTMC model under consideration. Let us assume that we have d running instances of the system, modeled by d statistically independent DTMCs, all starting from the same initial state s_0. Let us also assume that monitors collect *event traces*, for each instance, where each event represents the execution of an action of that instance, modeled as a transition of the DTMC. For example, in the case illustrated in Figure 3 the monitor collects the results of invocations—successes or failures—of $ExpressShipping$, as well as events from the $Profiler$, indicating access to the e-service application by returning users versus new customers. In practice, it may suffice to monitor environment phenomena that can change at run time. For simplicity and generality, however, we assume that traces contain events corresponding to all DTMC transitions of the application's model. Thus we do not distinguish between transitions representing phenomena that can change and transitions representing phenomena that do not or cannot change.

Let $\pi^{(h)}$ be the event trace for the h-th running instance and let π be the set of all traces for all the d running instances. From $\pi^{(h)}$, it is easy to compute the number of times $N_{i,j}^{(h)}$ each transition from state s_i to state s_j is taken during execution by instance h and then the total number $N_{ij} = \sum_h N_{ij}^{(h)}$ for all instances. The Bayesian technique consists of a formal, iterative process by which a-priori assumptions and experts' knowledge can be combined with real and updated data from monitors to produce an a-posteriori model, which is more adherent to the current state of the environment. The principle of the Bayesian estimation lays on Bayes' theorem, by which we can combine prior model and monitors' data as follows:

$$f(\mathbf{P}|\pi) = \frac{f(\pi|\mathbf{P}) \cdot f(\mathbf{P})}{f(\pi)} \tag{2}$$

where $f(\mathbf{P})$ is the current information about $\mathbf{P}$, namely its prior probability density function and π is the set of all traces for all the d running instances of the system collected during the observation period that eventually leads to the update.

In Equation (2), $f(\mathbf{P}|\pi)$ is the posterior density function for $\mathbf{P}$ updated from π. Notice that both the prior and the posterior, being probability density functions, contain a characterization of all our knowledge about $\mathbf{P}$, including our uncertainty. We use this information as estimator of the actual value of $\mathbf{P}$ and collect data in order to update prior knowledge and refine our estimations. We will later describe the analytical form of prior and posterior distributions of $\mathbf{P}$. The *likelihood function* $f(\pi|\mathbf{P})$ summarizes the new knowledge gathered through the sample π. The likelihood function expresses the probability of observing the sample set π given the prior knowledge about $\mathbf{P}$, thus it is in practice a measure of goodness of the current information in the prior.

Let us proceed with the estimation process. In Bayesian jargon, the *prior* probability is the *unconditional* probability of $\mathbf{P}$ while the *posterior* is the *conditional* probability given sample and priori information [27]. The Bayesian process allows us to combine the two available sources of information: our current knowledge about $\mathbf{P}$, embedded in the prior, and the new information contained in the sample π coming from monitors. As prior distribution for $\mathbf{P}$ we use the Dirichlet distribution [27].

Consider a row p_i of the matrix $\mathbf{P}$. First of all, it can be assumed that p_i does not depend on other rows of $\mathbf{P}$ because of the Markov hypothesis. Its entries p_{ij} represent the probability of moving toward state s_j given that the Markov process is in state s_i. It can be intuitively seen as a multinomial distribution, i.e. one with a finite discrete set of possible outcomes, over the n states of the DTMC that can be reached from state s_i with probability, respectively, $p_{i0}, p_{i1}, ..., p_{in}$ [27]. The Dirichlet distribution is a conjugate prior of the parameters of such a multinomial distribution; this means that the posterior is also characterized by a Dirichlet distribution.

Let $\mathbf{X} = (x_0, x_1, ..., x_n)$ be random vector whose entries represent the fraction of times the process moves from state s_i to each state s_j among all the possible n destination states, the Dirichlet probability density function is defined as follows (in terms of the Gamma function $\Gamma(\cdot)$ [27]):

$$\mathbf{X} \sim Dir(\alpha \cdot p_{i0}, \alpha \cdot p_{i1}, ..., \alpha \cdot p_{in}) = \frac{\Gamma(\sum_{j=0}^{n} \alpha \cdot p_{ij})}{\prod_{j=0}^{n} \Gamma(\alpha \cdot p_{ij})} \prod_{j=0}^{n} x_j^{\alpha \cdot p_{ij} - 1} \tag{3}$$

where $\alpha \in \mathbb{R}_{>0}$ is the *mass parameter*, which expresses the strength of the prior knowledge: a higher initial value for α means higher confidence on the original assumptions about the environment. They are also called "prior observation counts" in the sense that their value reflects the confidence on the prior "as if it was observed α times". Notice that giving higher confidence to the prior makes the estimation more robust with respect to outlier samples, but may result in a slower convergence of the estimator to the actual value.

The prior density function for each row of $\mathbf{P}$ at the set up of the system is a Dirichlet distribution with p_{ij} equal to the initially assumed values for transition probabilities and where the value for α is chosen based on the initial confidence one has on the correctness of the assumed values.

When a sample π is produced by monitors, we update the prior based on the newly gathered information. The update procedure is based on the likelihood of the sample π and leads to the definition of the posterior distribution. The likelihood $f(\pi|\mathbf{P})$ can be computed as follows:

$$Pr(\pi|\mathbf{P}) = Pr(\{\pi^{(1)}, \pi^{(2)}, ..., \pi^{(\mathbf{d})}\}|\mathbf{P}) = \prod_{h=1}^{d} \prod_{i,j} p_{i,j}^{N_{i,j}^{(h)}} \tag{4}$$

That is, given the assumed statistical independence among the d running instances, the joint probability of their observation is the product of the probabilities of observing each of them. In addition, the probability of observing each trace, given the Markov assumption, is the probability of observing each transition the number of times it occurred, no matter the order in which it occurs.

According to [55], by combining (3) and (4) in (2) through simple mathematical steps we obtain that the posterior is a Dirichlet distribution with the updated values for α' and p'_{ij} defined as follows (let $N_i = \sum_j N_{ij}$):

$$\begin{cases} \alpha' = \alpha + N_i \\ p'_{i,j} = \frac{\alpha}{\alpha'} \cdot p_{i,j} \ + \ \frac{N_{i,j}}{\alpha'} \end{cases} \tag{5}$$

The estimation, described by Equation (5), which computes the updated values for the entries of matrix $\mathbf{P}$, is iteratively performed for each incoming sample π; i.e., the posterior computed at step i becomes the prior at step $i + 1$, keeping the model continuously updated. Notice that the update rule is quite simple and fast to compute.

Concerning the role of α, by looking at equation (5) it is clear that by taking $\alpha \to 0$ in the initial step of the iterative algorithm the prior becomes non-informative, in the sense that the estimated distribution is just given by the empirical data observed in the monitored sample

The same happens when $N_i \to \infty$, that is, after a large amount of data is gathered by monitors. The Bayesian estimator converges to the frequentist *Maximum Likelihood Estimator* (MLE) (namely $N_{i,j}/N_i$). However, through the Bayesian approach we are able to embed design-time assumptions coming from past experience and then refine them to get closer to the observed changing reality. In the frequentist approach there is no way to take these assumptions into consideration. Convergence of the estimated model to a realistic one can thus be slower, due to the lack of an informative initial model.

The described method that continuously updates the DTMC can be combined with a method described in [23], which can recognize change-points (i.e., significant discontinuities in the behavior of the environment). Once a change-point is recognized, it is possible to derive initial estimates for the values α that are subsequently updated with the approach described here. This approach is further discussed in Section 6.2.

5.2. Model Retrofitting with Performance Parameters

In this section we present a Bayesian estimator for the exit rate of each state of a CTMC, which exploits the formalization of CTMCs presented in Section 3.3 and the estimation of the entries of matrix $\mathbf{P}$ of the embedded DTMC through the Bayesian method exposed in the previous section.

As already introduced in Section 3.3, the residence time x in each state s is modeled through an exponential distribution having *exit rate* λ[5]. Our purpose is to define an estimator for λ[6]. The probability density function of the Exponential distribution is:

$$f(x|\lambda) = \lambda e^{-\lambda x} \qquad \text{if } x \geq 0, \text{ 0 otherwise.} \tag{6}$$

Assuming that k is the number of elements in the sample set x, the Likelihood function for the Exponential distribution in (6) is:

$$\begin{aligned} &Pr(x|\lambda) = \prod_{j=1}^{k} \lambda e^{-\lambda x_j} = \lambda^k e^{-\lambda k \bar{x}} \\ &\text{where } \bar{x} = \frac{1}{k}\sum_{j=1}^{k} x_j \end{aligned} \tag{7}$$

For the Exponential distribution, a convenient conjugate prior is the Gamma distribution $\Gamma(\alpha, \beta)$[7] having the following probability density function, expressed in terms of the Gamma function $\Gamma(\cdot)$ [27]:

$$f(\lambda|\alpha,\beta) = \Gamma(\lambda;\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda}$$

As in Section 5.1, the prior represents knowledge about λ_i before data is observed and the posterior represents knowledge about the parameter after data is observed. The Γ distribution is a correct estimator for λ [27], thus its expected value is the estimation we are looking for:

$$\lambda = E[f(\lambda|\alpha,\beta)] = \frac{\alpha}{\beta} \tag{8}$$

Being the Gamma distribution a conjugate prior, the posterior distribution will belong to the same family and it can be constructed as follows (to shorten the construction process, we use the relation $\propto$ instead of $=$, meaning that the left term equals the right term up to a constant coefficient):

$$f(\lambda|x) \propto Pr(x|\lambda) \cdot f(\lambda|\alpha,\beta) = \lambda^n e^{-\lambda k\bar{x}} \cdot \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda} \propto \lambda^{(\alpha+k)-1}e^{-\lambda(\beta+k\bar{x})} \tag{9}$$

[5] Notice that the exit rate from a state depends only on that state, according to the Markov property for CTMCs.
[6] To improve readability, we refer in our treatment to a generic state s and a generic rate λ. Of course, the Bayesian method infers updated values for the rates of all states s_i.
[7] In probability theory, α, β are commonly called *shape* and *rate* of the Gamma distribution, respectively.

By normalizing Equation (9), (i.e., identifying the constant coefficient making its integral equal to one) it is possible to rewrite the posterior in a compact form as [27]:

$$f(\lambda|x) = \Gamma(\lambda; \alpha + k, \beta + k\bar{x}) \tag{10}$$

Hence, by recalling Equation (8), after the update of the prior with samples in x, the posterior leads to the update value for λ:

$$\lambda' = E[\Gamma(\lambda; \alpha + k, \beta + k\bar{x})] = \frac{\alpha + k}{\beta + k\bar{x}} \tag{11}$$

Let us now make some considerations on the roles of parameters α and β. By looking at Equation (10), α can be seen as a counter of the number of observed samples, while β is the sum of the sampled values, that is, the sum of the measured residence times in state s. The role of α is thus analogous to the case of Section 5.1 and it will be exploited hereafter to manage the confidence on the initial assumptions in a similar way.

Before discussing the setup of the estimation process, by looking at Equations (8) and (11) we can notice that there is a relation between λ, α and β, which is preserved through the entire estimation process. In order to produce a compact and meaningful update rule we can make use of such relation noticing that, after each iteration, $\beta = \alpha/\lambda$. Hence, the value of β is univocally determined from α and λ, and this allows us to express the update rule with respect to α and λ only, as follows:

$$\begin{cases} \alpha' = \alpha + k \\ \lambda' = \frac{\lambda\alpha'}{\alpha + \lambda k\bar{x}} \end{cases} \tag{12}$$

Given the update rule in (12), we now need to assign the initial values for α and λ. The initial value for λ is the one assumed at design time. Concerning α, a high initial value corresponds to an high confidence on the initial assumption and, as in Section 5.1, its values has to be intended “as if λ has been observed α times”.

Similar considerations to the ones made in Section 5.1 concerning the relation between the Bayesian estimator and the MLE one can be restated here. In particular, $\alpha \to 0$ means that the initial assumption tends to be ignored. Hence, from Equation (12), we obtain $\lambda = 1/\bar{x}$, i.e., the MLE [58]. Also, the Bayesian estimator converges to the MLE as the number of samples growths, that is, when the empirical information significantly overcomes the initial guess in the iterative process:

$$\lim_{k\to\infty} E(f(\lambda|\alpha + k, \beta + k\bar{x})) = \lim_{k\to\infty} \frac{\alpha + k}{\beta + k\bar{x}} = \frac{1}{\bar{x}}$$

For the same reasons expressed in Section 5.1, even if after a long time the Bayesian and the Maximum Likelihood converge to the same value, even though, in general, the Bayesian approach can provide a better estimate, since it enables to embed prior knowledge.

6. Run-Time Verification in Action

In this section we show how retrofitted models can be continuously analyzed at runtime. We provide examples that mostly refer to reliability analysis. The approach applies to performance along the same lines. To support the approach, we developed the KAMI environment for run-time verification, in which the model checker PRISM is integrated together with components that perform Bayesian inference. Run-time data are collected by external monitors that can be plugged into the KAMI environment.

KAMI allows the inference procedure to be tuned by selecting, for each analyzed sample, the values for the parameters that weigh the trustworthiness of the initial guess (the prior) as opposed to the measured values provided by monitors, as explained in Sections 5.1 and 5.2. We refer to these parameters as *inference parameters* in the sequel.

In the sequel we first show how continuous verification supports fault detection, which may possibly lead to failure detection and failure prediction. We also provide some experimental results about the effectiveness of the retrofitting method and show how it can be improved. The discussion follows the case study introduced in Section 4.

6.1. Fault Detection, Failure Prediction, and Failure Detection

We observed earlier that our design-time assumptions and initial high-level description of the overall system's architecture allow us to achieve a successful verification against the reliability and performance requirements. The proof relies on the assumption that assertions D hold at run time. However, the run-time behavior of the environment may diverge from the design-time assumptions.

As a first example, let us assume that the probability of failure of *ExpShipping* is actually slightly higher than the assumed value for verification, e.g. equal to 0.067 ($D_{s,4} = 0.067$). In this case, it can be easily proved by running the model checker that requirement $R2$ is violated, since the probability of an *ExpShipping* failure for a returning customer is equal to 0.042, which is greater than 0.035. Let us see how this problem is actually discovered through our run-time verification procedure in KAMI. A monitor collects run-time data from the running instances of the system as described in [22] and the Bayesian approach described earlier, for given values of inference parameters, produces estimates for the transition probabilities of a posterior DTMC, which represents a more accurate model of environment phenomena. In particular, assuming that the monitor has observed a trace of events containing 600 invocations to the *ExpShipping* operation with 40 failing invocations, the Bayesian estimator produces a new estimate for $D_{s,4}$ equal to 0.06. Although the "actual" failure probability (0.067) has not been identified yet, the inferred value is sufficient for KAMI to discover that requirement $R2$ is violated.

Similarly, a domain property $D_{u,i}$ related with user profile could differ at run time from what we expected at design time, leading to a requirements violation. For example, let us assume now that the probability of failure of *ExpShipping* is equal to what we assumed at design time. Instead, the probability that a returning customer chooses express shipping changes to 0.633 (i.e., $D_{u,2} = 0.633$). In this case, requirement $R2$ is also violated, since the probability of failure of *ExpShipping* for a returning customer is equal to 0.04, which is greater then 0.035. Let us again observe what KAMI actually may do to discover the problem. Assuming a trace of monitored events where 380 out of 600 users recognized as returning customers choose to do express shipping, the Bayesian estimator produces a new estimate for $D_{u,2}$ equal to 0.6. Although the "actual" probability that a returning customer chooses *ExpShipping* (0.633) has not been identified yet, the inferred value is sufficient for KAMI to discover that requirement $R2$ is violated.

In the previous examples we have been able to discover deviations (from the values of $D_{u,2}$ and $D_{s,4}$) that lead to a violation of the requirements. In particular, the higher failure probability of *ExpShipping* is considered as a *failure* of the outsourcer, because it violates its reliability contract. The violation of requirement $R2$ in the previous two cases indicates that KAMI reveals a *fault detection* for the e-commerce application. The fault may become visible to our stakeholders (i.e., it leads to a failure) or it simply predicts that sooner or later the fault will become visible (i.e., it does failure prediction).

Let us now focus on the first example above in which $R2$ is violated because of a failure of the external *ExpShipping* service, which violates its contract, formally represented by $D_{s,4}$. Notice that stakeholders of the e-commerce application, on behalf of whom we must guarantee probabilistic requirements R, perceive a failure by comparing the number of successful invocations to the system with respect to a (sufficiently large) number of total invocations. For example, in the specific case of $R2$, returning customers may simply count the number of failed requests for express shipping ($RC_{exp,f}$) out of the total number of requests (RC_{exp}) in a specific interval of time. Consequently, $R2$ is perceived as violated if:

$$\frac{RC_{exp,f}}{RC_{exp}} \geq 0.035 \tag{13}$$

Let us consider again a trace that contains 600 invocations to *ExpShipping*, out of which 40 fail. Suppose that 100 out of the 600 invocations of the sample are issued by returning customers. Let x be the number of failed *ExpShipping* invocations out of the 40. If $x < 35$ then KAMI identifies a fault, but no failure is perceived yet by the stakeholders, since condition (5) does not hold. The detected fault, however, corresponds to a failure prediction, since sooner or later—if no further changes occur—the violation of requirement $R2$ will be perceived, when more returning customers will show up (remember that the expected percentage of returning customers in a sufficiently large population is 35%!). Instead in the case where $35 \leq x \leq 40$ the fault is perceived by the stakeholders, i.e., we achieve failure detection.

6.2. Empirical Evaluation

In this section we provide some experimental results about our method. The results are obtained via simulation in Matlab [33]. We show how the two Bayesian estimators proposed in Section 5 can be used to update inaccurate design-time assumptions.

We first show two examples, concerning reliability and performance, respectively. Each sample for reliability estimation has been extracted by random generation from a Bernoulli test. The parameter of such distribution is to be intended as the failure probability of the observed operation. Samples for performance estimation have been obtained by the default exponential random sample generator of Matlab. The parameter to estimate is the exit rate from an observed state. The estimators proposed in Section 5 are able to identify the real values of observed parameters, even in case of a wrong a priori assumption[8]. Also, they prove to be resilient against outliers and spikes. Consider, for example, Figure 5, which shows how a reliability parameter (the failure probability of the *ExpShipping* operation) converges from the initial design-time value to the real value that characterizes the run-time environment. The diagram also shows that a short sudden deviation from the "normal" behavior (a spike) does not affect the estimator significantly. A similar behavior is shown in Figure 6 in the case of a performance parameter. This example refers to the rate of a CTMC node that was assumed to have value 20 at design time, but instead turns out to have value 10.

Figure 7 shows what happens at run time if a further change occurs later in the value of real parameter. The example refers to reliability. It is clear from the diagram that convergence is slow, since the whole history of past data up the change point negatively affects the speed of convergence. A significant improvement of the approach can be achieved if we split the inference in two steps:

1. we identify the coordinate of the point at which the law that governs the behavior of the environment phenomenon changes (in the example, 50000);
2. we re-apply our inference method from the identified change point, using as prior the last estimated value.

The result of this approach is shown in Figure 8. In this case, convergence is very efficient. This method, which relies on change-point detection, was proposed in [23].

7. Related Work

The work described in this paper covers several areas of research that are currently quite active internationally. It is rooted at large into software evolution, a research area that was started in the 1970s by M. Lehman and others [8, 50, 51]. However it departs radically from what mainstream software evolution research has been focusing on over the years, since our goal is to achieve on-line self-adaptation of the software to changes in the environment in which it is embedded. This is largely motivated by the scenarios discussed in [21]. The idea that systems should be capable of reconfiguring themselves automatically to respond to changes in the requirements and in the environment has been proposed by the field of *autonomic computing* [42, 37] and has since spurred research within the software community. Research is currently quite active at all levels: from languages, to middleware and architecture, formal methods, verification, deployment. A view of the state of the art can be found in [59, 18]. Our work is an attempt to provide sound foundations—based on formal methods—to dynamically adaptive software systems that must comply with probabilistic non-functional requirements.

A number of interesting and related research approaches can be found in the area of service-oriented systems, for which dynamic evolution and adaptation is the norm. Some approaches share our emphasis on verification; e.g., [9, 61, 1]. Our approach differs because we emphasize *continuous* verification. In fact, we strongly believe that self-adaptation must be based on continuous verification and that continuous verification—in turn—requires models to be run and analyzed at run time. Two research communities became active in recent years to focus on these two areas ([52, 11]). The reader may refer to the conference series on run-time verification[9], which originated in 2001 and the workshop series on models at run

[8] For simplicity here we ignore the problem of choosing the values of the inference parameters in formula (3). We uniformly use the value 1. For additional comments, see [22].
[9] http://www.runtime-verification.org/

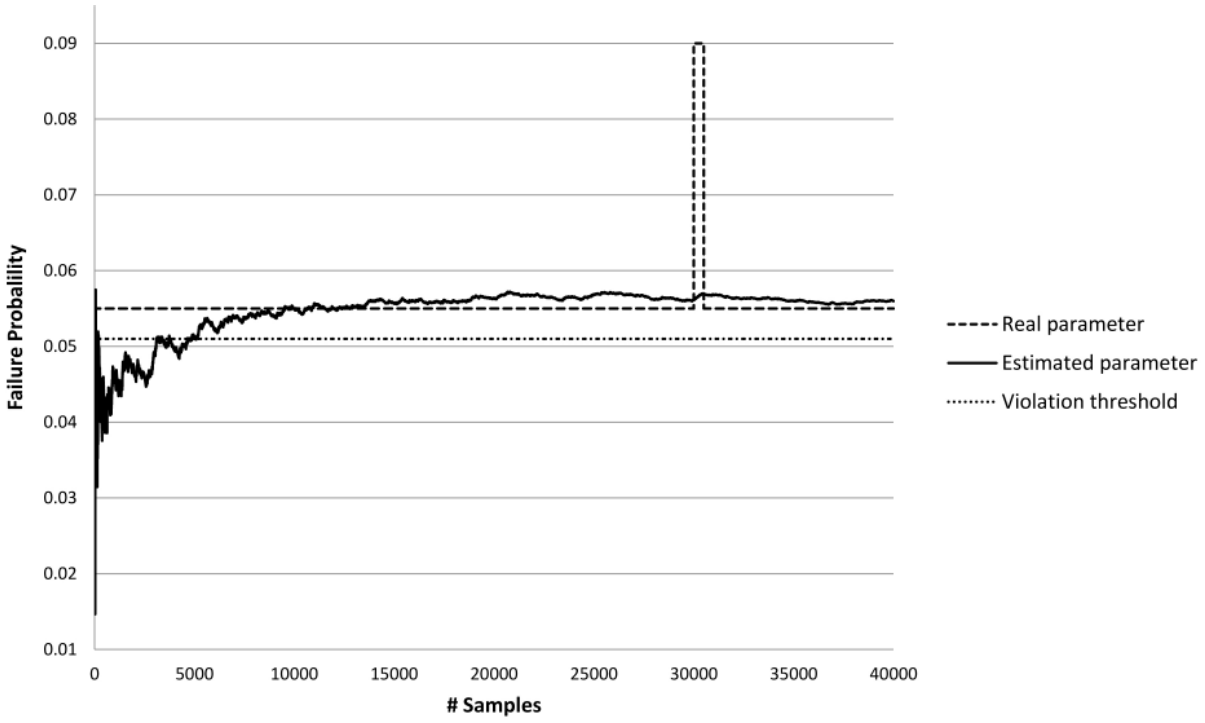


Fig. 5. Failure probability estimation for the *ExpShipping* operation.

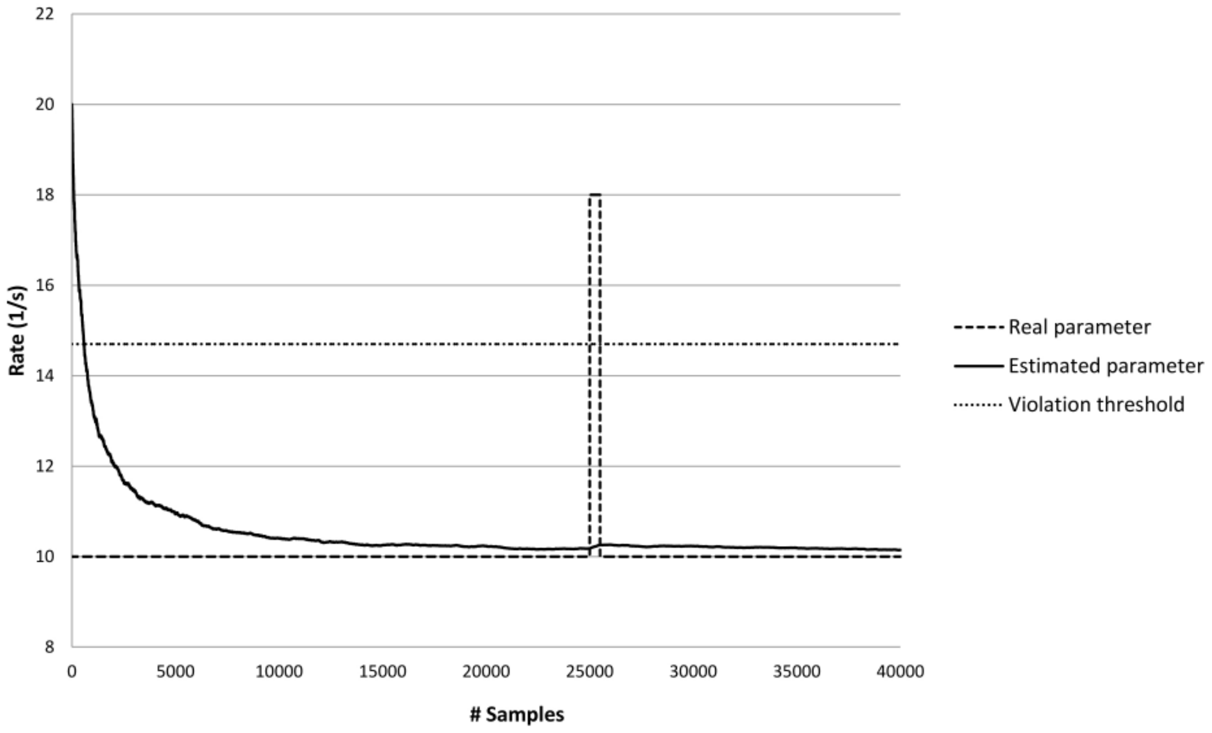


Fig. 6. Exit rate estimation for the returning users' interface.

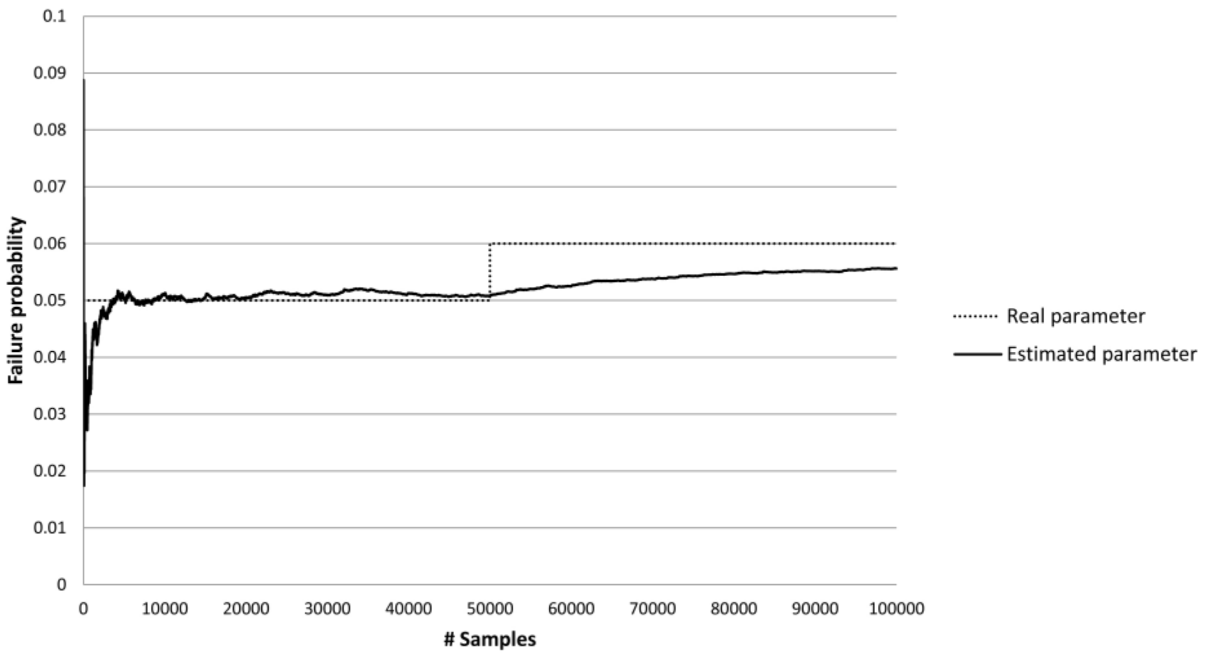


Fig. 7. Effect of a further change

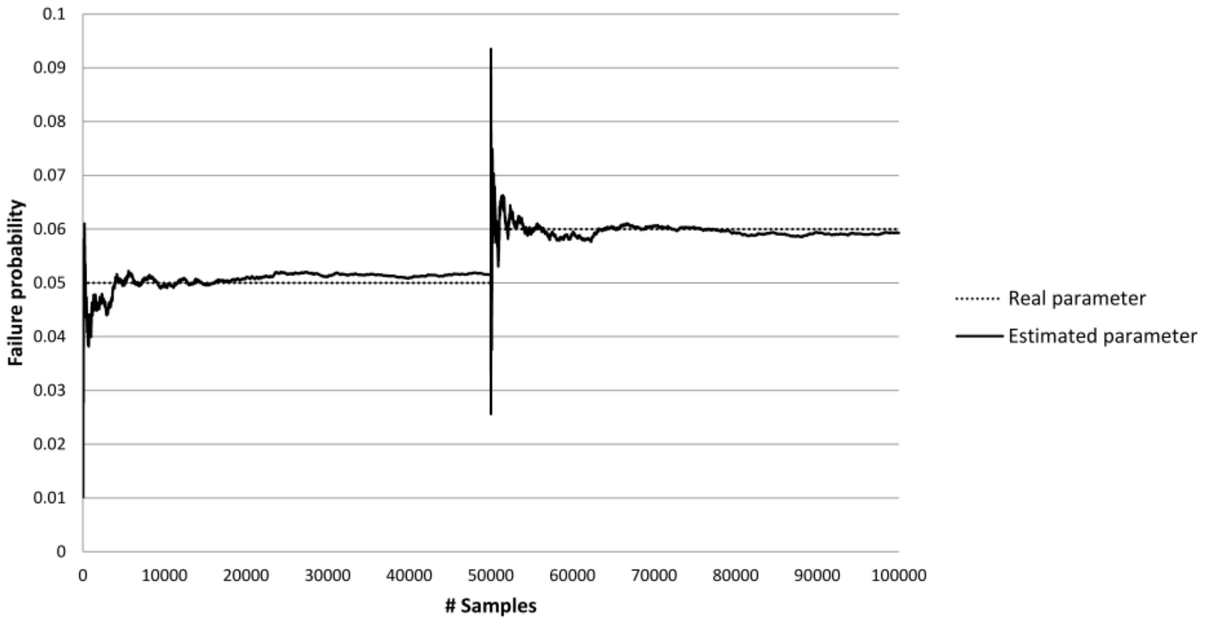


Fig. 8. Inference based on change-point detection

time[10], which exists since 2006, to have an idea of the main approaches being investigated and the results accomplished. The research approach described in [66] is also relevant in our case. The work we presented in this paper follows from very similar motivations but has a rather different focus. The authors of [66] are mainly concerned with requirements formalized by means of linear temporal logic (LTL) and evaluated on Petri Net models of the software's behavior. In this framework it is possible to express liveness properties, constraints on the occurrences of specific events, and invariants on the entire execution. Its application is mainly concerned with satisfaction of correctness assertions. It cannot model stochastic behaviors nor can it express (and verify) probabilistic properties.

The approach described in [16] also uses a formal model and quantitative analysis to drive adaptation. However, our approach differs because we support the automatic update of the model through parameter inference.

As for the models on which our approach is founded, we use Markov models, for which we adopt the verification aids provided by the stochastic model checking research community [49, 40, 5]. In particular, we have mostly used PRISM [36, 47], and also MRMC [41].

8. Conclusions and Future Work

The work we presented here provides a comprehensive view of the KAMI approach we developed to support the level of continuous verification that is needed to ensure that dynamic (self)adaptation to environmental changes can be founded on a solid formal background. The focus of KAMI is on non-functional properties that can be specified quantitatively in a probabilistic way. In particular, it focuses on reliability and performance properties. KAMI is implemented as a distributed framework with a plugin architecture, which allows new tools to be incorporated to support other modeling notations and analysis procedures.

The main distinctive features of the approach are:

- It is based on a general reference framework for change and evolution, cast into the Jackson-Zave approach to requirements.
- Self-adaptation to environmental changes that might disrupt satisfaction of non-functional requirements—our ultimate goal—is grounded on formal verification.
- Verification is performed at the model level, and requires models to be kept alive at run time.
- Monitors collect real-world data and machine learning approaches retrofit detected environment changes as updated model parameters.

The proposed approach provides a holistic view of adaptive systems that spans from development time of an application to its run time evolution. The paper illustrates the approach through the running example of a simplified e-commerce application. We presented both a coherent conceptual view of the proposed approach and some experimental data, which show the feasibility of the proposed approach to model parameter update.

The results achieved so fare are quite promising. However, a number of further improvements and several important research questions remain open, and are part of our research agenda. We briefly list some of the most prominent issues.

1. The KAMI approach illustrated above is based on repeatedly running the model checker on the updated model at run time, to detect possible requirements violations. However, running traditional model checking algorithms at run time may be incompatible, in terms of expected response time from the tool, with the timing requirements needed for the reaction, if a preventive or a repairing action is required. If the problem is discovered too late, it may be hard to react meaningfully through self-adaptation. We believe that more research is needed to devise strategies for run-time efficient model checking. Some steps in this direction have been already be achieved ([34, 48, 25]), but more can be done.
2. An alternative approach to the one presented in the paper could investigate the use of interval estimators instead of single-value. This might lead to the adoption of more expressive models, like *Interval Markov Chains*[44], where intervals model knowledge inaccuracies.
3. So far in our work we focused on detection of the need for self-adaptation. The next important step is the selection of a self-adaptation strategy. This is an area in which research is quite active internationally, and

[10] http://www.comp.lancs.ac.uk/~bencomo/MRT/

several approaches are currently investigated (see [18, 46]) to devise strategies for implementing changes in the model and in the implementation as the need for self-adaptation is recognized. Most proposals, including some work developed in our group ([17, 28]), are still quite preliminary and hard to assess and compare. No approach seems to be sufficiently general and scalability is often an issue. An intriguing approach to self-adaptation would be to apply the feed-back approach of control theory to dynamically tuning the software. Initial work in this area can be found in [35, 43, 53], but no generalized approach has ever been attempted.

4. After the need for a reconfiguration of the application is recognized as a consequence of the choice of an adaptation strategy, changes must be installed in the running application. As opposed to conventional software changes, which occur off-line and require re-deployment of the application, in this context we need dynamic reconfigurations where certain components are deployed and bound to the application as it is running. This requires revisiting and enhancing previous work on dynamic reconfiguration ([45, 63]).
5. Adaptation to other requirements violations (both functional and non-functional) should be supported. Increasingly important non-functional requirements, for example, deal with energy consumption, and energy-aware adaptation appears to be of high practical relevance. Markov models with rewards may provide an adequate modeling support to reasoning about energy consumption, and model checkers provide support for verification. Incorporation of such models and verifiers in the continuous adaptation framework should be investigated.
6. We assumed here that the initial model of the application and of the environment (S and D, according to the used terminology) are given. More should be done on the requirements elicitation phase, through which we should be able to derive an understanding of environment changes, which should guide the development of the architectural variants that can lead to the dynamic reconfigurations needed to reify the dynamic adaptations.
7. Finally, an important research direction concerns the way humans can interact with adaptive systems. For example, we assumed here that the properties of interest, which must be preserved through self-adaptation, are expressed formally using PCTL or CSL. Because software engineers may not be expected to be proficient in temporal logics, it may be useful to investigate the use of requirements patterns and of structured natural language as a bridge towards formal specification, along the lines proposed by [32][11]. Another area to explore is how to interact with a human in the loop whenever the system discovers that no self-adaptation strategy can be found to respond to certain detected environment changes. A manual change must be requested in this case, but the designer should be given access to all the collected data to reason about the problem and and to support a careful evolution of the application's design and implementation to deal with them.

Acknowledgments

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom. We are indebted to Raffaela Mirandola and Ilenia Epifani for their contributions during certain stages of this work.

References

[1] J. Abreu, F. Mazzanti, J. L. Fiadeiro, and S. Gnesi. A model-checking approach for service component architectures. In *FMOODS/FORTE*, pages 219–224, 2009.

[2] A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, Sterling, D. König, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0. OASIS Committee Draft, May 2006.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

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

[11] A proposal to integrate ProProST with KAMI and with the GPAC (General-Purpose Autonomic Computing) framework [14] has been recently investigated in [15].

[5] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
[6] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.
[7] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The spec# programming system: Challenges and directions. In *VSTTE*, pages 144–152, 2005.
[8] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.
[9] A. Bertolino, W. Emmerich, P. Inverardi, V. Issarny, F. K. Liotopoulos, and P. Plaza. Plastic: Providing lightweight & adaptable service technology for pervasive information & communication. In *ASE Workshops*, pages 65–70, 2008.
[10] J. Bézivin. Model driven engineering: An emerging technical space. In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 36–64. Springer, 2006.
[11] G. Blair, N. Bencomo, and R. France. Models@ run.time. *Computer*, 42(10):22 –27, 2009.
[12] BPEL. http://www.oasis-open.org/.
[13] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7:212–232, June 2005.
[14] R. Calinescu. General-purpose autonomic computing. In Y. Zhang, L. T. Yang, and M. K. Denko, editors, *Autonomic Computing and Networking*, pages 3–30. Springer US, 2009.
[15] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimisation in service-based systems—to appear. *IEEE Trans. Softw. Eng.*
[16] R. Calinescu and M. Kwiatkowska. Using quantitative analysis to implement autonomic it systems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 100–110, May 2009.
[17] L. Cavallaro, E. Di Nitto, P. Pelliccione, M. Pradella, and M. Tivoli. Synthesizing adapters for conversational web-services from their wsdl interface. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 104–113, New York, NY, USA, 2010. ACM.
[18] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009.
[19] R. C. Cheung. A user-oriented software reliability model. *IEEE Trans. Softw. Eng.*, 6(2):118–125, 1980.
[20] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
[21] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A Journey to Highly Dynamic, Self-adaptive Service-based Applications. *Automated Software Engineering.*, 15(3-4):313–341, 2008.
[22] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model Evolution by Run-Time Adaptation. In *Proceedings of the 31st International Conference on Software Engineering*, pages 111–121. IEEE Computer Society, 2009.
[23] I. Epifani, C. Ghezzi, and G. Tamburrelli. Change-point detection for black-box services. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 227–236, New York, NY, USA, 2010. ACM.
[24] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110, 2010.
[25] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011—to appear.
[26] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In *QoSA '08: Proceedings of the 4th International Conference on the Quality of Software Architectures*, Karlsruhe, Germany, 2008.
[27] A. Gelman. *Bayesian data analysis*. CRC press, 2004.
[28] C. Ghezzi, A. Motta, V. Panzica La Manna, and G. Tamburrelli. Qos driven dynamic binding in-the-many. *Sixth International Conference on the Quality of Software Architectures*, QoSA 2010.
[29] C. Ghezzi and G. Tamburrelli. Predicting performance properties for open systems with KAMI. In *QoSA '09: Proceedings of the 5th International Conference on the Quality of Software Architectures*, pages 70–85, Berlin, Heidelberg, 2009. Springer-Verlag.
[30] C. Ghezzi and G. Tamburrelli. Reasoning on Non-Functional Requirements for Integrated Services. In *Proceedings of the 17th International Requirements Engineering Conference*, pages 69–78. IEEE Computer Society, 2009.
[31] K. Goseva-Popstojanova and K. S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179 – 204, 2001.
[32] L. Grunske. Specification patterns for probabilistic quality properties. In Robby, editor, *ICSE*, pages 31–40. ACM, 2008.
[33] M. Guide. The mathworks. *Inc., Natick, MA*, 5, 1998.
[34] E. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric markov models. *Model Checking Software*, pages 88–106, 2009.
[35] J. L. Hellerstein. Self-managing systems: A control theory foundation. *Local Computer Networks, Annual IEEE Conference on*, 0:708–708, 2004.
[36] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS06)*, 3920:441–444, 2006.
[37] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.
[38] A. Immonen and E. Niemela. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.

[39] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 15–24, New York, NY, USA, 1995. ACM.

[40] J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic ctmc model checking. In L. de Alfaro and S. Gilmore, editors, *Process Algebra and Probabilistic Methods. Performance Modelling and Verification*, volume 2165 of *Lecture Notes in Computer Science*, pages 23–38. Springer Berlin / Heidelberg, 2001.

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

[42] J. Kephart and D. Chess. The Vision of Autonomic Computing. *COMPUTER*, pages 41–50, 2003.

[43] M. Kihl, A. Robertsson, M. Andersson, and B. Wittenmark. Control-theoretic analysis of admission control mechanisms for web server systems. *The World Wide Web Journal, Springer*, 11(1-2008):93–116, Aug. 2007.

[44] I. O. Kozine and L. V. Utkin. Interval-valued finite markov chains. *Reliable Computing*, 8:97–113, 2002.

[45] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16:1293–1306, November 1990.

[46] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering*, pages 259–268, 2007.

[47] M. Kwiatkowska, G. Norman, and D. Parker. Prism 2.0: a tool for probabilistic model checking. *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 322–323, 2004.

[48] M. Kwiatkowska, D. Parker, and H. Qu. Incremental quantitative verification for markov decision processes. *unpublished—submitted for publication*, 2011.

[49] M. Z. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *SFM*, pages 220–270, 2007.

[50] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[51] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change.* Academic Press Professional, Inc., 1985.

[52] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[53] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *CDC*, pages 3736–3741, 2010.

[54] B. Meyer. Contract-driven development. In *FASE*, page 11, 2007.

[55] K. Ng, G. Tian, and M. Tang. *Dirichlet and Related Distributions: Theory, Methods and Applications.* Wiley Series in Probability and Statistics. John Wiley & Sons, 2011.

[56] OpenID. http://www.openid.com/.

[57] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques.* John Wiley & Sons, 2005.

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

[59] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2), 2009.

[60] J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *In: Proc. of 26th Intl. Conference on Software Engineering (ICSE*, pages 179–188. IEEE Press, 2004.

[61] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In *FMICS*, pages 133–148, 2007.

[62] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* J. Wiley and Sons, 2009.

[63] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Software Eng.*, 33(12):856–868, 2007.

[64] WSDL. http://www.w3.org/2002/ws/desc/.

[65] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.

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