

Conquering Complexity via Seamless Integration of Design-Time and Run-Time Verification

Antonio Filieri, Carlo Ghezzi, Raffaella Mirandola, and Giordano Tamburrelli

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

Abstract. The complexity of modern software systems has grown enormously in the past years with users always demanding for new features and better quality of service. Software applications evolved not only in terms of size, but also in the criticality of the services supported. At the same time, software artifacts changed from being monolithic and centralized to modular, distributed, and dynamic. Systems are now composed of heterogeneous components and infrastructures on which software is configured and deployed. Interactions with the external environment and the structure of the application, in terms of components and interconnections, are often required to change dynamically. All these causes challenge our ability to achieve acceptable levels of dependability. To guarantee system dependability, it is necessary to combine off-line (development-time) analysis techniques with run-time mechanisms for continuous verification. Off-line verification checks the correct behavior of the various components of the application under given assumptions on the embedding environment. But because verification can be incomplete, the assumptions about reality it relies upon are subject to uncertainty and variability and, in addition, the various parts of a complex system may evolve independently, it is necessary to extend verification to also cope with the runtime behavior of software. This paper motivates the need for continuous verification to guarantee dependability and shows how this goal may be tackled. In particular, it focuses attention on two important dependability attributes: reliability and performance.

1 Introduction

Software is the driving engine of modern society. Most human activities—including critical ones—are either software enabled or entirely managed by software. Examples range from healthcare and transportation to commerce and manufacturing to entertainment and education. As software is becoming ubiquitous and society increasingly relies on it, the adverse impact of unreliable or unpredictable software cannot be tolerated. Software systems are required to be *dependable*, to avoid damaging effects that can range from loss of business to loss of human life.

At the same time, the complexity of modern software systems has grown enormously in the past years with users always demanding for new features

and better quality of service. Software systems changed from being monolithic and centralized to modular, distributed, and dynamic. They are increasingly composed of heterogeneous components and infrastructures on which software is configured and deployed. When an application is initially designed, software engineers often only have a partial and incomplete knowledge of the external environment in which the application will be embedded at run time. Design may therefore be subject to high uncertainty. This is further exacerbated by the fact that the structure of the application, in terms of components and inter-connections, often changes dynamically. New components may become available and published by providers for use by potential clients. Some components may disappear, or become obsolete, and new ones may be discovered dynamically. This may happen, for example, in the case of Web service-based systems [8]. This also happens in pervasive computing scenarios where devices that run application components are mobile [56]. Because of mobility, and more generally context change, certain components may become unreachable, while others become visible during the application's lifetime. Finally, requirements also change continuously and unpredictably, in a way that is hard to anticipate when systems are initially built. Because of uncertainty and continuous external changes the software application is subject to continuous adaptation and evolution. All this is challenging our ability to achieve the required levels of dependability. F.P. Brooks anticipated this when he said *Complexity is the business we are in and complexity is what limits us* [16].

This paper focuses on how to manage design-time uncertainty and run-time changes and how to verify that the software evolves dynamically without disrupting the dependability of applications. We refer to *dependability* as broadly defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance¹ as:

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

Dependability thus includes as special cases such attributes as reliability, availability, performance, safety, security. In this paper we focus our attention on two main dependability requirements that typically arise in the case of decentralized and distributed applications: namely, *reliability* and *performance*. Both reliability and performance depend on environment conditions that are hard to predict at design time, and are subject to a high degree of uncertainty. For example, performance may depend on end-user profiles, on network congestion, on load conditions of external services that are integrated in the application. Similarly, reliability may depend on the behavior of the network and of the external services that compose the application being built.

Our approach to the development and operation of complex and dynamically evolvable software systems is rooted in the use of formal models. Hereafter we discuss how uncertainty and anticipation of future changes can be taken into account when the system is initially designed. In particular, we focus on the formal

¹ <http://www.dependability.org/wg10.4/>

models that can be built at design time to support an initial assessment that the application satisfies the requirements. We also show that models should be kept alive at run time and continuously verified to check that the changes with respect to the design-time assumptions do not bring to requirements violations. This requires seamless integration of design-time and run-time verification. If requirements violations are detected, appropriate actions must be undertaken, ranging from off-line evolution to on-line adaptation. In particular, much research is currently investigating the extent to which the software can respond to predicted or detected requirements violation through self-managed reactions, in an autonomic manner. These, however, are out of the scope of this paper, which only focuses on design-time and run-time verification.

Our contribution is structured as follows. Section 2 introduces a running example, which is inspired by a Web-service based *e-Health* application, called TeleAssistance (TA). Section 3 surveys the main formal notations we use to model applications and reason about compliance of its design with respect to its non-functional requirements. We then discuss (Section 4) how design-time requirements verification may be accomplished in presence of uncertainty. This will be done by first providing high-level models of the running TA example and then by formally verifying requirements satisfaction under some assumptions about the run-time environment in which TA will be embedded. Section 5 focuses on monitoring the run-time behavior and performing continuous run-time verification. Finally, Section 7 provides pointers to on-going work and draws some conclusions.

2 A Running Example

This section illustrates the running example adopted in this paper to illustrate the proposed design and run-time approach. An e-Health application, initially studied in [7] and then further used as a case-study in [28], is designed as a distributed system for medical assistance. The application is built by composing a number of existing Web services. Web-service compositions (and service-oriented architectures in general [26]) make an excellent case for the need of keeping models alive at run time. A Web-service composition is an orchestration of Web services aimed at building a new service by exploiting a set of existing ones. The orchestration is performed through the BPEL workflow language [2]. A BPEL composition is, in turn, a service that can be composed with other services in a recursive manner. BPEL instances coordinate services that are typically managed by independent organizations, other than the owner of the service composition. This distributed ownership implies that the final functional and non-functional properties of the composed service rely on behaviors of third-party components that influence the obtained results, as we will discuss hereafter.

The running example, called TeleAssistance (TA), focuses on a composite service supporting remote assistance of patients who live in their homes. Figure 1 illustrates the TA composite service through a graphical notation into which BPEL constructs are mapped. The mapping between BPEL constructs and the

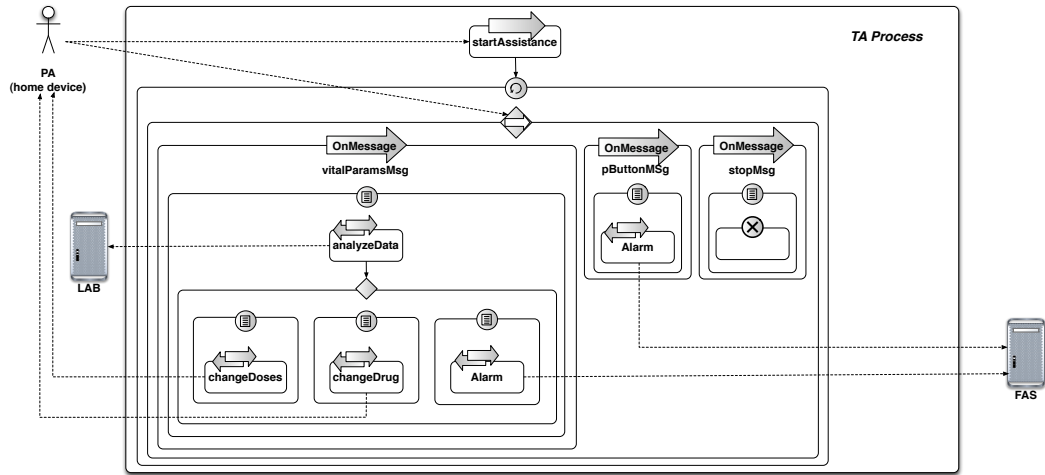


Fig. 1. TA BPEL Process

corresponding graphical notation is described in Table 1. The reader who ignores BPEL may find a brief summary in the Appendix.

Table 1. BPEL Graphical Notation

Activity	Notation	Activity	Notation	Activity	Notation
<i>receive</i>		<i>wait</i>		<i>pick</i>	
<i>invoke</i>		<i>terminate</i>		<i>flow</i>	
<i>reply</i>		<i>sequence</i>		<i>fault handler</i>	
<i>assign</i>		<i>switch</i>		<i>event handler</i>	
<i>throw</i>		<i>while</i>		<i>compensation handler</i>	

The process starts as soon as a Patient (PA) enables the home device supplied by TA, which sends a message to the process' receive activity `startAssistance`. Then, it enters an infinite loop: every iteration is a pick activity that suspends the execution and waits for one of the following three messages: (1) `vitalParamsMsg`, (2) `pButtonMsg`, or (3) `stopMsg`. The first message contains the patient's vital parameters that are forwarded by the BPEL process to the Medical Laboratory service (LAB) by invoking the operation `analyzeData`. The LAB is in charge of analyzing the data and replies by sending a result value stored in a variable

analysisResult. A field of the variable contains a value that can be: *changeDrug*, *changeDoses* or *sendAlarm*. The latter message triggers the intervention of a First-Aid Squad (FAS) composed of doctors, nurses, and paramedics, whose task is to visit the patient at home in case of emergency. To alert the squad, the TA process invokes the operation alarm of the FAS. The message *pButtonMsg* caused by pressing a panic button also generates an alarm sent to the FAS. Furthermore, the message *stopMsg* indicates that the patient may decide to cancel the TA service.

The system should be designed to satisfy a number of requirements concerning the Quality of Service (QoS), among which:

- **R1:** *The probability $P1$ that no failures ever occurs is greater than 0.7*
- **R2:** *If a *changeDrug* or a *changeDoses* has occurred the probability $P2$ that the next message received by the TA generates an alarm which fails (i.e., the FAS is not notified) is less than 0.015*
- **R3:** *Assuming that alarms generated by *pButtonMsg* have lower priority than the alarms generated by *analyzeData*, the probability $P3$ that a high priority alarm fails (i.e., it is not notified to the FAS) is less than 0.012*
- **R4:** *The average response time of the Alarm service (RT_{Alarm}) must be less than 1 second*
- **R5:** *The utilization of the AnalyzeData ($U_{AnalyzeData}$) must be less than 90%*
- **R6:** *The average number of pending requests (i.e., the queue length) to the FAS service (QL_{FAS}) must be less than 60*

Notice that requirements $R1 - 3$ refer to the system's reliability. Conversely, $R4 - 6$ refer to performance.

In the sequel, we will discuss how formal models can support design-time verification that the system being designed satisfies the requirements, under certain assumptions about the behavior of the environment. We will then show how the models can be kept alive at run time to support continuous verification that requirements are not violated despite changes in the assumptions under which the system was initially verified.

3 Non Functional Models for Complex Systems

This section provides an introduction to the non-functional models we adopt to express QoS properties. As previously introduced, we focus on reliability and performance. As non-functional models we rely respectively on Discrete Time Markov Chains (DTMCs) and Queueing Networks (QNs). Let us first introduce Markov models in general and then describe DTMCs and QNs [15].

3.1 Markov Models.

Several approaches exist in the literature for model-based quality analysis and prediction, spanning the use of stochastic Petri nets, queueing networks, layered

queueing network, stochastic process algebras, Markov processes, fault trees, statistical models and simulation models (see [3] for a recent review and classification of models for software quality analysis).

In this work, we focus on Markov models, which are a very general evaluation model that can be used to reason about performance and reliability properties. Furthermore, Markov models include other modeling approaches as special cases, such as queueing networks, stochastic Petri nets [59] and stochastic process algebras [24].

Specifically, Markov models are stochastic processes defined as state-transition systems augmented with probabilities. Formally, a stochastic process is a collection of random variables $X(t), t \in T$ all defined on a common sample (probability) space. $X(t)$ is the state at time t , where t is a value in a set T that can be either discrete or continuous. In Markov models, states represent possible configurations of the system being modeled. Transitions between states occur at discrete or continuous time-steps and the probability of making transitions is given by exponential probability distributions. The Markov property characterizes these models: it means that, given the present state, future states are independent of the past. In other words, the description of the present state fully captures all the information that could influence the future evolution of the process. The most used Markov models include:

- *Discrete Time Markov Chains* (DTMCs), which are the simplest Markovian model where transitions between states happen at discrete time steps;
- *Continuous Time Markov Chains* (CTMCs), where the value associated with each outgoing transition from a state is intended not as a probability but as a parameter of an exponential probability distribution (transition rate);
- *Markov Decision Processes* (MDPs) [63], which are an extension of DTMCs allowing multiple probabilistic behaviors to be specified as output of a state. These behaviors are selected non-deterministically. MDPs are characterized by a discrete set of states representing possible configurations of the system being modeled and transitions between states occur in discrete time-steps, but in each state there is also a non-deterministic choice between several discrete probability distributions over successor states.

The solution of Markovian models aims at determining the system behavior as time t approaches infinity. It consists of the evaluation of the stationary probability π_s of each state s of the model.

The analytical solution techniques for Markov models differ according to the specific model and to the underlying assumptions (e.g., transient or non-transient states, continuous vs. discrete time, etc.). For example, the evaluation of the stationary probability π_s of a DTMC model requires the solution of a linear system whose size is given by the number of states. The exact solution of such a system can be obtained only when the number of states is finite or when the matrix of transition probabilities has a specific form. DTMCs including transient and absorbing states necessitate a more complex analysis for the evaluation of the average number of visits and absorbing probabilities. The detailed derivation is discussed in [15]. A problem of Markov models, which also

similar evaluation models face, is the explosion of the number of states when they are used to model real systems [15]. To tackle this problem tool support (e.g., PRISM [52]) with efficient symbolic representations and state space reduction techniques [45,53] like partial-order reduction, bisimulation-based lumping and symmetry reduction are required.

Given a Markov model it is possible to represent QoS requirements as non-ambiguous properties expressed in an appropriate logic, such as probabilistic temporal logics PCTL (Probabilistic Computation Tree Logic) [40], PCTL* [4], PTCTL (Probabilistic Timed CTL) [54] and CSL (Continuous Stochastic Logic) [5]. The significant benefits of using logic-based requirements specifications include the ability to define these requirements concisely and unambiguously, and to analyze them using rigorous, mathematically-based tools such as model checkers. Furthermore, for logic-based specification-formalism the correct definition of QoS properties is supported with specification patterns [27,38,37,49] and structured English grammars [38,49,71].

Markov models are widely used at design time to derive performance and/or reliability metrics. For example, the work presented in [36] discusses in depth the problem of modeling and analyzing the reliability of service-based applications and presents a method for the reliability prediction of service compositions based on the analysis of the implied Markovian models. The analysis of a CTMC implied by a BPEL process is also used in [67] as a way to derive performance and reliability indices of a service composition.

Discrete Time Markov Chains. DTMCs are specifically used to model reliability concerns. As introduced before, DTMCs are defined as state-transition systems augmented with probabilities. *States* represent possible configurations of the system. *Transitions* among states occur at discrete time and have an associated probability. DTMCs are discrete stochastic processes with the Markov property, according to which the probability distributions of future states depend only upon the current state.

Formally, a (labeled) DTMC is tuple (S, s_0, P, L) where

- S is a finite set of states
- $S_0 \subseteq S$ is a set of initial states
- $P : S \times S \rightarrow [0, 1]$ is a stochastic transition matrix ($\sum_{s' \in S} P(s, s') = 1 \quad \forall s \in S$). An element $P(s_i, s_j)$ represents the probability that the next state of the process will be s_j given that the current state is s_i .
- $L : S \rightarrow 2^{AP}$ is a labeling function which assigns to each state the set of *Atomic Propositions* $a \subseteq AP$ holding in s . As discussed in [51], AP formally is a fixed, finite set of atomic propositions used to label states with the properties of interest which can be verified by a stochastic model checker.

A DTMC evolves from the initial state by executing a transition at each discrete time instant. Being at time i in a state s , at time $i+1$ the model will be in s' with probability $P(s, s')$. The transition can take place only if $P(s, s') > 0$.

A state $s \in S$ is said to be an *absorbing state* if $P(s, s) = 1$. If a DTMC contains at least one absorbing state, the DTMC itself is said to be an *absorbing*

DTMC. Furthermore, we assume that every state in the DTMC is reachable from the initial state, that is there exists at least a sequence of transitions from the initial state to every other state.

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

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

where I is an r by r identity matrix, 0 is an r by t zero matrix, R is a nonzero t by r matrix and Q is a t by t matrix.

Consider now two distinct transient states s_i and s_j . The probability of moving from s_i to s_j in exactly 2 steps is $\sum_{s_x \in S} P(s_i, s_x) * P(s_x, s_j)$. Generalizing the process for a k -steps path and recalling the definition of matrix product, it comes out that the probability of moving from any transient state s_i to any other transient state s_j in exactly k steps corresponds to the entry (s_i, s_j) of the matrix Q^k . By generalization, the probability of moving from s_i to s_j in 0 steps is 1 iff $s_i = s_j$, that is Q^0 .

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

In the simplest model for reliability analysis, the DTMC modeling a task will have two absorbing states, one representing the correct accomplishment of the task, the other representing the failure of the system. The use of absorbing states is commonly extended to represent different failures. For example, a failure state may be associated with each invocation of an external service that may fail. A basic feature of a reasoning system in this framework is to provide an estimate for the probability of reaching an absorbing state or the ability to state whether the probability of reaching an absorbing state associated with a failure is less than a certain threshold.

3.2 Queueing Networks.

For performance models we exploit QNs. As introduced before, they can be reduced to a Markov model and the solution of a QN might be obtained by solving the underlying Markov process. However, for some classes of QNs, efficient analytical solution techniques exist to determine the average values of the performance metrics (e.g., average response time, utilization, etc.) or, in some cases, also the percentile distribution of the metric of interest.

QNs [15,55] are a widely adopted modeling technique for performance analysis. QNs are composed by a finite set of: (1) *Service Centers*, (2) *Links*, (3) *Sources and Sinks*, and (4) *Delay Centers*.

Service centers model system resources that process customer request. Each service center is composed of a *Server* and a *Queue*. Queues can be characterized by a finite or an infinite length. In this work we focus on service centers with infinite queues. Service centers are connected through *Links* that form the network

topology. Servers process *jobs*—hereafter we refer to requests interchangeably with the term *jobs*—retrieved from their queue following a specific policy (e.g., FIFO). Each processed request is then routed to another service center through connections provided by links. More precisely, each server, contained in every service center, picks the next job from its queue (if not empty), processes it, and selects one link that routes the processed request to the queue of another service center. It is possible to specify a policy for link selection (e.g., probabilistic, round robin, etc.). The time spent in every server by each request is modeled by continuous distributions such as exponential or Poisson distributions. Jobs are generated by source nodes connected with links to the rest of the QN. Source nodes are also characterized by continuous time distributions that model request inter-arrival times. Sink nodes represent the points where job leave the system. Finally, delay centers are nodes of the QN connected with links to the rest of the network exactly as service centers, but they do not have an associated queue. Delay centers are described only by a service time, with a continuous distribution, without an associated queue. They correspond to service centers with infinite servers.

After modeling a software system as a queueing network, the model has to be evaluated in order to determine quantitative performance metrics, such as:

- *Utilization*: the ratio between the server’s busy time over the total time.
- *Response Time*: the interval between submission of a request into the QN and output of results.
- *Queue Length*: the average queue length for a given service center.
- *Throughput*: the number of requests processed per unit of time.

The above measures are defined for a single service center, but they can also apply to the whole network. A first step in the evaluation of a QN can be achieved by determining the system bounds; specifically, upper and lower bounds on system throughput and response time can be computed as functions of the system workload intensity (number or arrival rate of customers). Bounds usually require a very little computational effort, especially for simple kinds of QN, like single-class networks [21,48].

More accurate results can be achieved by solving the equations which govern the QN behavior. Solution techniques can be broadly classified as *analytical methods* (which can be *exact* or *approximate*) and *simulation methods*. Analytical methods determine functional relations between model parameters and performance metrics. Queueing networks satisfying the BCMP theorem assumptions (see [10] for further details) are an important class of models also known as *product-form models*. Such models are the only ones that can be solved efficiently, while the solution time of the equations governing non-product-form queueing network grows exponentially with the size of the network. Hence, in practical situations the time required for the solution of non-product-form networks becomes prohibitive and approximate solutions have to be adopted. Analytical solutions often provide only the average values of the performance metrics (e.g., average response time, utilization, etc.). Detailed solutions can be obtained by solving the Markov process underlying the queueing network model (details about the

derivation of the Markov process can be found in [15]).

For non-product-form QNs very often simulation is used to evaluate performance metrics. Simulation is a very general and versatile technique to study the evolution of a software system, which is described by a simulation program that mimics the dynamic behavior of the system by representing its components and interactions in terms of functional relations. Non-functional attributes are estimated by evaluating the values of a set of observations gathered in the simulation runs. Simulation results are then obtained by performing statistical analyses of multiple runs [44]. With simulation it is possible to obtain very accurate results but at the cost of a higher computational effort with respect to the analytical solution of QNs.

Modeling Complex Systems with Queueing networks. In modern complex systems, components may fall into different categories. First, they may differ in the way they are used (*use mode*). Their use may be *exclusive*; that is, the component is only used by the currently designed application. In this case, the component may be modeled as a service center, since we have full control of the flows of requests into its input queue. In other cases, the component is *shared* among different applications, which we may not know, although they concurrently access it. The component cannot be modeled as service center because other jobs, which we cannot control, also can access the service. In such a case, the component can be more simply—but less accurately—modeled as a delay center.

As an example of these two cases, consider a component which provides functionalities for video encoding and decoding. In case it is a component-off-the-shelf (COTS), which is deployed within the current application and it is used exclusively by it, the designer has full control and visibility of its activations, and thus it can be modeled by a service center. If, however, the tool is offered by a provider as a Web service, it is potentially accessed by many clients, and the designer has no control nor visibility of the queues of requests.


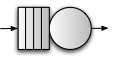
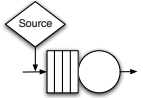

Another key factor that must be considered by the modeler is *visibility* of the internals of the component. Both accuracy and trust of the component's performance characteristics depend on how detailed the designer's knowledge is of the component's internals. If an accurate description of the component's architecture is available, its performance can be predicted quite accurately, for example using a design-time tool like Palladio [12]. If instead the component is a black-box, like in the case of Web services, the designer must rely on less trustable figures published by the service provider or inferred by past observations. Note that visibility is often related to *ownership*. If one owns a component, then normally one also has full access to its internals, and conversely. Furthermore, it is also related with *stability*. Whenever a component is owned, it only evolves under control of the owner. If an accurate model of the component is available, there is no need to monitor the component at run time for possible deviations and, consequently, to update the model.

The above discussion leads to the following main component categories:

- *White-Box (WB)* components. Their internal architecture is fully visible and understood by the designer; for example, they have been developed in-house. In addition, their use is exclusive by the current application.
- *Grey-Box (GB)* components. Their use is exclusive, but their internals are not known; only the executable version of the component is available. COTS are a typical example.
- *White-Box Shared (WBS)* components. The designer has full visibility of the component, which however is not used exclusively within the application being developed. An example is an in-house developed Web-service that is used by the current application, but is also exported for use by others.
- *Black-Box (BB)* components. The designer has no visibility of the internals of the component, whose use is shared with other unknown clients. An example is an externally developed Web service developed by third parties that is available on-line.

Table 2 summarizes the previous discussion by showing the main categories of components, the choices we made for modeling them via QNs, and the graphical notation we use.

Table 2. QN Notation for Open Systems

Notation	Name	Use Mode	Visibility	Description
	White-Box	exclusive	yes	service center
	Grey-Box	exclusive	no	service center
	White-Box Shared	shared	yes	service center with source node
	Black-Box	shared	no	delay center

4 Design-Time Modeling and Verification of the TA System

Hereafter we apply the formalisms discussed in the previous section in the initial design and verification of the TA system. The first step of our approach consists of developing models that can be used to reason about our non-functional properties of interest (reliability and performance). To do so, we identify the parts that are subject to uncertainty and which may change in the value of quality attributes. We especially focus on two major sources of uncertainty and

volatility: *user profiles*, which describe how system functions will be used by user transactions, and *external components (services)*, which may change their quality of service over time in an unexpected and uncontrolled manner. These may be viewed as black-box components, accessible via an abstract interface that only provides visibility of the stable information upon which we can rely.

We assume that uncertain information can be expressed in probabilistic terms. This may be difficult in practice, but it is a necessary step in our approach if we want to be able to predict and assess non-functional properties at design time. Several practical guidelines may be followed as a guidance through this step. For example, initial estimates may be provided by the designer based on past experience with similar systems. In the case where external components (services) managed by third parties are integrated into the current system, the estimate may be provided by the service-level agreement subscribed by the provider or by ad-hoc tests performed by client stubs.

The next section shows how we model reliability of the TA system via DTMCs. We then show how performance can be modeled by exploiting QNs. Finally we discuss how an initial assessment of requirements satisfaction may be obtained by analyzing the models.

4.1 DTMCs at work

Figure 2 illustrates the result obtained by modeling the TA running example introduced in Section 2. The modeling activity consists of identifying relevant states of the system, assigning probabilities to branches, and failure probabilities to service invocations. Notice that failure states are highlighted in grey. In this example, we adopted numerical values chosen for illustrative purposes; real-world medical applications usually require lower failure probabilities. Usage profiles are also represented in Figure 2 as probabilities associated with transitions. As an example, consider the transitions exiting state 0. With probability 0.3 the user pushes a button to generate an alarm, whose notification to the first-aid squad fails with probability 0.04.

The DTMC derivation can be done either manually or through automatic transformation techniques. Several contributions that appeared in the literature proposed techniques to derive DTMC starting from a formal description of the system's behavior (e.g.,[36,67,31]).

4.2 QNs at work

Figure 3 illustrates the result obtained by modeling the TA example with a QN. Notice that transition probabilities among service centers are consistent with values used in Figure 2. Before applying the concepts and the taxonomy illustrated in Section 3.2, we need to take into consideration some performance data describing the behavior of the components part of the TA system. Such data, as for transition probabilities in DTMCs, might be provided by domain experts or other existing systems. Table 3 summarizes this information set.

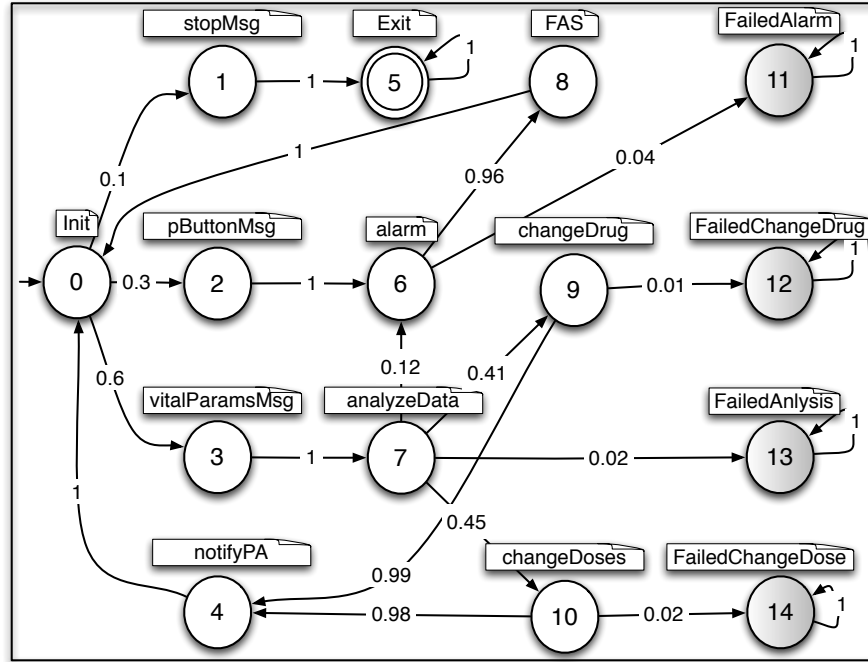


Fig. 2. TeleAssistance DTMC Model

Table 3. QN Additional Parameters

Component	Parameter	Value
Source 0	Arrival Rate	Exponential with $\lambda = 0.5$
Source 1	Arrival Rate	Exponential with $\lambda = 0.1$
startAssistance	Service Time	Exponential with $\lambda = 1$
startAssistance	Queue	∞
stopMsg	Service Time	Exponential with $\lambda = 1$
stopMsg	Queue	∞
FAS	Service Time	Exponential with $\lambda = 1.45$
FAS	Queue	∞
Alarm	Service Time	Exponential with $\lambda = 1.5$
AnalyzeData	Service Time	Exponential with $\lambda = 2.5$
changeDoses	Service Time	Exponential with $\lambda = 1.2$
changeDrug	Service Time	Exponential with $\lambda = 1.2$

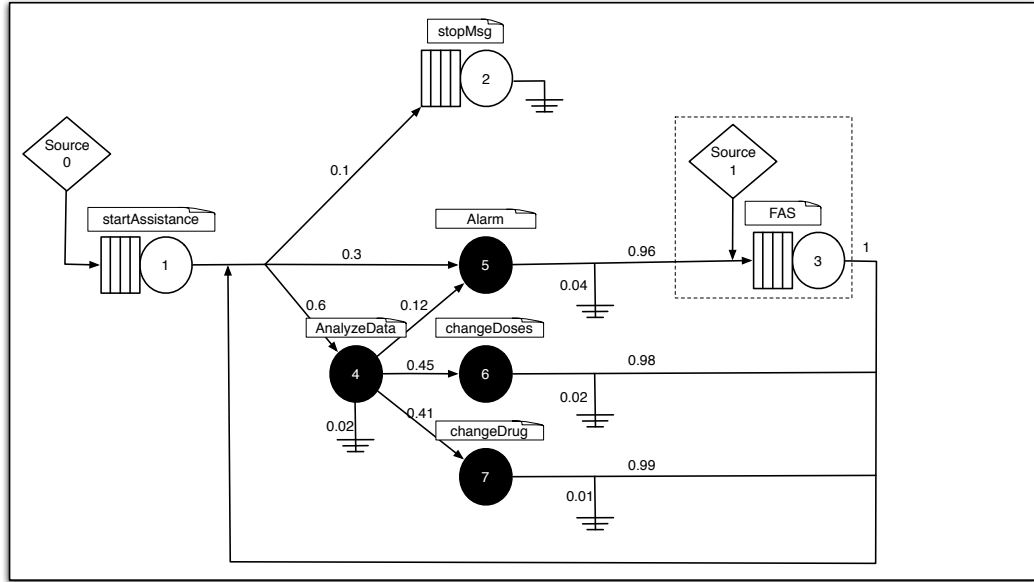


Fig. 3. TeleAssistance QN Model

Concerning the components and services part of the TA system, we assumed the changeDrug, changeDoses and sendAlarm service centers as Web services provided by third-party organizations. Conversely, we considered the FAS service as a service owned by the same organization of the TA system but potentially used by other healthcare functions. According to these assumptions, we model the former as Black-box centers and the latter as White-Box Shared.

To facilitate the software engineer's task, several methodologies can be found in the literature to support transformation techniques that can derive QN-based models (both product and non-product) starting from software models. Some of the proposed methods are reviewed in [6,11,1].

4.3 Design-Time Verification

Once the models of the application under design are available, they can be analyzed to verify requirements satisfaction. Let us start our discussion from reliability and let us consider the reliability model illustrated in Figure 2.

The reliability requirements $R1 - R3$ can be proven to hold for the composite service. Several instruments are in place to verify stochastic properties on DTMCs, with different pros and cons. The most basic approach is based on probability theory's formulas, which can be solved by means of numerical methods [65]; these approaches are typically fast and very accurate, but most often used for simple properties such as the probability of reaching a certain state. The

most popular verification tools nowadays are the probabilistic model checkers. The two most widely adopted are PRISM [42] and MRMC [46]. Model-checkers come with logics expressive enough to assert complex properties over the set of all possible paths through a DTMC, but they typically make use of iterative methods which provide a finite accuracy (though this can be arbitrarily high at the price of a polynomially longer computation time). For very large systems which are hard to be analyzed by means of mathematical methods, it is also possible to apply some verifiers which adopt Monte-Carlo simulation [68].

For example, by using the DTMC probabilistic model checker PRISM [42], we obtain: $P1=0.7421$, $P2=0.0147$, $P3=0.0048$. As we discussed earlier, model parameters (i.e., transition probabilities) might be provided by: (1) domain experts, (2) similar existing systems, or (3) previous versions of the system under design. In any case, such parameters represent only estimates and run-time analyses are in charge of refining them together with a continuous verification of the compliance with the system’s requirements as illustrated later on in Section 5.

Let us now consider performance and the QN model illustrated in Figure 3. By relying on parameters listed in Table 3 and exploiting a QN solver such as JMT [14] we can evaluate the performance requirements $R4 - R6$ and prove to hold for the composite service. In particular, we obtained: $RT_{Alarm}=0.6667$, $U_{AnalyzeData}=0.8906$, $QL_{FAS}=53.4771$. Notice that model parameters might be retrieved as previously mentioned for DTMCs transition probabilities.

5 Supporting Run-Time Verification

After an application is developed, it is deployed in the target environment to interact with the real world. Regrettably, at run time reality may subvert the assumptions made by software engineers at design time. For example, user profiles may differ from the expected values, or may later change during operation. Likewise, the performance of an external service integrated in the application may change, due for example to the deployment of a new version of the service which provides additional features. Similarly, a service’s reliability may unexpectedly decrease, due to the upload of a new, buggier release. For these reasons, it is necessary that verification continues after the application’s delivery, to check if changes cause a violation of requirements. If they do, the application must also change.

We distinguish between two kinds of change: adaptation and evolution [56]. *Adaptation* refers to the actions taken at run time and affecting the architectural level, to react to the changing environment in which the application operates. In fact, changes in the physical context may often require the software architecture to also change. As an example, a certain service used by the application may become unaccessible as a new physical context is entered during execution. Conversely, a new service may become visible. It may also happen that a certain service is changed unexpectedly by its owner and the change is found to be incompatible with its use from the current application. *Evolution* instead refers to changes in the application that are the consequence of changes in the require-

ments. For example, a new feature is added to the TA system to support medical diagnosis remotely via video interaction with the patient. Adaptation must be increasingly supported in an autonomic way. We use the term self-adaptation in this case.

In our approach, evolution and (self) adaptation are triggered by run-time verification, whenever a requirements failure is detected [35]. To support run-time verification, the application’s model has to be alive at run time and it must be fed with updated values of the parameters, which reflect the detected changes in environment conditions. To detect changes—in turn—suitable run-time monitors must be activated to collect the relevant data from the environment [33,9]. In the TA example, a monitor should detect changes in the usage profiles and in the reliability and performance characteristics of the external services. To do so, the data observed at run time must be converted into probabilities that are used to annotate the DTMC and the QN models of the TA example. The conversion can be performed by learning algorithms, typically based on a Bayesian approach, as shown in [28].

In the TA example, let us consider the effect of the following situations that may be occur at run time.

- The service providing the FAS functionality is discontinued for some time (it fails with probability 1). The verification procedure for reliability requirements (a probabilistic model checker) detects a run-time violation of requirement R1.
- The *notifyPA* operation, which was supposed to be completely reliable (failure probability equal to 0) is found out to fail with probability 0.01. The model checker in this case detects a run-time violation of requirement R2.
- The distribution of reactions to *analyzeData* is found to be quite different from the one assumed at design time. The probability discovered at run-time that *changeDoses* is diagnosed is 0.20 instead of 0.45, the probability that *changeDrug* is diagnosed is 0.31 instead of 0.41, the probability that *alarm* is generated is 0.47 instead of 0.12, while the probability that a failure is experienced has exactly the value hypothesized by the designer (0.20). The model checker detects a run-time violation of requirement R3.

Performance requirements can also be checked for possible violation at run time. Hereafter we provide a few examples of cases where the environment’s behavior differs from the assumptions made during design and this would lead to requirements violations, detected by the QN analyzer:

- Assuming that due to contextual issues the alarm is able to answer to an average of 0.9 requests per second, instead of the value 1.5 expected, the average response time of the service grows from 0.667 to 1.111, violating requirement R4.
- If from monitoring data the actual measured request processing rate of the data analyzer is 2 requests/sec, slower than the value expected at runtime (2.5), then the utilization of the analyzer becomes 1.1135. Such a value, being larger than 90%, leads to the violation of the requirement R5.

- The FAS component is shared with third parties. Their usage of the component is modeled by source1. If those entities, out from the control of the Tele Assistance company, increase their request rate from 0.1 to 0.2 requests per second, the waiting queue of the FAS saturates (in this model, the number of enqueued requests grows indefinitely) and begins to loose incoming requests. This violates requirement R6 because QL_{FAS} tends to ∞ .

6 Related Work

In the last years, QoS prediction has been extensively studied in the context of *traditional software systems*. In particular, there has been much interest in model transformation methodologies for the generation of analysis-oriented target models (including performance and reliability models) starting from design-oriented source models, possibly augmented with suitable annotations. Several proposals have been presented concerning the direct generation of performance analysis models. Each of these proposals focuses on a particular type of source design-oriented model and a particular type of target analysis-oriented model, with the former spanning UML, Message Sequence Chart, Use Case Maps, formal language as AEmilia, ADL languages such as Acme, and the latter spanning Petri nets, queueing networks, layered queueing network, stochastic process algebras, Markov processes (see [6] for a thorough overview of these proposals and the WOSP conference series [1] for recent proposals on this topic). A systematization of the current approaches in the framework of MDD and interesting insights on future trends on this topic can be found in [3]. Some proposals have also been presented for the generation of reliability models. All the proposals we are aware of start from UML models with proper annotations, and generate reliability models such as fault trees, state diagrams, Markov processes, hazard analysis techniques and Bayesian models (see [43,13] for a recent update on this topic).

More recently, with the increasing interest in the topic of reconfigurable and self-adaptive computing systems [23], several papers appeared in the literature dealing with self-adaptation of software systems to guarantee the fulfillment of QoS requirements. Hereafter, we present a short summary of existing work that makes use of models to perform this step. GPAC (General-Purpose Autonomic Computing), for example, is a tool-supported methodology for the model-driven development of self-managing IT systems [17]. The core component of GPAC is a generic autonomic manager capable of augmenting existing IT systems with a MAPE [47] autonomic computing loop. The GPAC tools and the probabilistic model checker PRISM [42] are used together successfully to develop autonomic systems involving dynamic power management and adaptive allocation of data-center resources [18]. KAMI [28] is another framework for model evolution by runtime parameter adaptation. KAMI focuses on Discrete Time Markov Chain models that are used to reason about non-functional properties of the system. The authors adapt the QoS properties of the model using Bayesian estimations based on runtime information, and the updated model allows the verification of

QoS requirements. The approach presented in [64] considers the QoS properties of a system in a web-service environment. The authors provide a language called SLAng, which allows the specification of QoS to be monitored.

The Models@Run.Time approach [62] proposes to leverage software models and to extend the applicability of model-driven engineering techniques to the runtime environment to enhance systems with dynamic adapting capabilities. In [69], the authors use an architecture-based approach to support dynamic adaptation. Rainbow [32] also updates architectural models to detect inconsistencies and in this way it is able to correct certain types of faults. A different use of models at runtime for system adaptation is taken in [58]. The authors update the model based on execution traces of the system. In [73] the authors describe a methodology for estimation of model parameters through Kalman filtering. This work is based on a continuous monitoring that provides run-time data feeding a Kalman filter, aimed at updating the performance model.

In [66], the authors propose a conceptual model dealing with changes in dynamic software evolution. Besides, they apply this model to a simple case study, in order to evaluate the effectiveness of fine-grained adaptation changes like service-level degrading/upgrading action considering also the possibility to perform actions involving the overall resource management. The approach proposed in [60] deals with QoS-based reconfigurations at design time. The authors propose a method based on evolutionary algorithms where different design alternatives are automatically generated and evaluated for different quality attributes. In this way, the software architect is provided with a decision making tool enabling the selection of the design alternatives that best fits multiple quality objectives. Menascé et al. [61] developed the SASSY framework for generating service-oriented architectures based on quality requirements. Based on an initial model of the required service types and their communication, SASSY generates an optimal architecture by selecting the best services and potentially adding patterns such as replication or load balancing. In [57] an approach for performance-aware reconfiguration of degradable software systems called PARSY (Performance Aware Reconfiguration of software SYstems) is presented. PARSY tunes individual components in order to maximize the system utility with the constraint of keeping the system response time below a pre defined threshold. PARSY uses a closed Queueing Network model to select the components to upgrade or degrade.

In the area of service-based systems (SBS), devising QoS-driven adaptation methodologies is of utmost importance in the envisaged dynamic environment in which they operate. Most of the proposed methodologies for QoS-driven adaptation of SBS address this problem as a service selection problem (e.g., [25,19,72]). Other papers have instead considered service-based adaptation through workflow restructuring, exploiting the inherent redundancy of SBS (e.g., [30,39,41].) In [20] a unified framework is proposed where service selection is integrated with other kinds of workflow restructuring, to achieve a greater flexibility in the adaptation.

7 Conclusions and Future Work

In this paper we focus on complex, evolvable, and adaptable software applications that live in highly dynamic environments and yet need to provide service in a dependable manner. These requirements affect the way software is designed and operated at run time. The most striking consequence is that models should be kept alive at run time to support a verification activity that extends to run time.

We envision three important directions for future work. First, it is important to investigate how detected requirements violations at run time may drive self-adaptation, to achieve autonomic behavior. In our research group, we achieved some preliminary results for restricted cases of self-adaptation in [22,34], but much remains to be done.

Another important research direction should investigate the methods that fit the specific requirements of run-time verification. In this paper, we assumed that the same verification procedures that are used at design time can also be used at run time. This is of course often an unrealistic assumption. Because run-time reactions that lead to self-adaptation are triggered by failures in requirements verification, the time consumed by the verification procedure must be compatible with the time limits within which a reaction must take place. The model checkers available for requirements verification are not designed for on-line use, but rather to explore design-time tradeoffs. Efficient verification algorithms need to be developed to fully support run-time verification. An initial step in this direction is explored in [29].

A third research direction in which we are currently engaging concerns the mechanisms that must support the run-time reconfigurations that are produced as a result of self-adaptation. Dynamic reconfiguration must occur dynamically, as the application is running and providing service. The goal is to preserve correctness and at the same time perform the change in a timely manner, without disrupting the service. Our work is focusing on extending previous work by Kramer and Magee [50], which was further extended by [70].

Appendix: BPEL Overview

BPEL, *Business Process Execution Language*, is an XML-based workflow language conceived for the definition and the execution of service compositions. BPEL processes comprise variables, with different visibility levels, and the workflow logic expressed as a composition of elementary activities. Activities comprise tasks like: *Receive*, *Invoke*, and *Reply* that are related to the interaction with other services. Moreover it is possible to perform assignments (*Assign*), throwing exceptions (*Throw*), pausing (*Wait*) or stopping the process (*Terminate*).

Branch, loop, while, sequence and switch constraints manage the control flow of BPEL processes. The pick construct is peculiar to the domain of concurrent and distributed systems, and waits for the first out of several incoming messages, or timer alarms to occur, to execute the activities associated with such an event. Each scope may contain the definition of the several handlers: (1) an Event

Handler that reacts to an event by executing a specific activity, (2) a Fault Handler catches faults in the local scope, and (3) a Compensation Handler aimed at restoring the effects of a previously unsuccessful transaction. For a complete description of BPEL language see [2,?]. The graphical representation used in this paper is described earlier in Section 2.

Acknowledgments

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

1. Wosp : Proceedings of the international workshop on software and performance, 1998-2008.
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. D. Ardagna, C. Ghezzi, and R. Mirandola. Rethinking the use of models in software architecture. In *4th International Conference on the Quality of Software Architectures, QoSA 2008*, volume 5281 of *LNCS*, pages 1–27. Springer, 2008.
4. A Aziz, V. Singhal, and F. Balarin. It usually works: The temporal logic of stochastic systems. In Pierre Wolper, editor, *Proc. 7th International Conference on Computer Aided Verification, CAV 95*, volume 939 of *LNCS*, pages 155–165. Springer, 1995.
5. C. Baier, J-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In Jos C. M. Baeten and Sjouke Mauw, editors, *Proc. 10th International Conference on Concurrency Theory, CONCUR 99*, volume 1664 of *LNCS*, pages 146–161. Springer, 1999.
6. S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.
7. L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, December 2007.
8. L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.
9. L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *Proceedings of the 2nd international conference on Service oriented computing, ICSOC '04*, pages 193–202, New York, NY, USA, 2004. ACM.
10. F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, April 1975.
11. S. Becker, L. Grunske, R. Mirandola, and S. Overhage. Performance prediction of component-based systems - a survey from an engineering perspective. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 2006.

12. S. Becker, H. Koziolok, and R. Reussner. Model-based performance prediction with the palladio component model. In *WOSP '07: Proceedings of the 6th International Workshop on Software and Performance*, pages 54–65, New York, NY, USA, 2007. ACM.
13. S. Bernardi, J. Merseguer, and D.D. Petriu. Adding dependability analysis capabilities to the MARTE profile. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings, MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 736–750. Springer, 2008.
14. M. Bertoli, G. Casale, and G. Serazzi. The jmt simulator for performance evaluation of non-product-form queueing networks. In *Annual Simulation Symposium*, pages 3–10, Norfolk, VA, US, 2007. IEEE Computer Society.
15. G. Bolch, S. Greiner, H. de Meer, and K.S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience New York, NY, USA, 1998.
16. F.P. Brooks. *The mythical man-month: Essays on software engineering*. Pearson Education, 1975.
17. R. Calinescu. General-purpose autonomic computing. In Mieso K. Denko, Laurence Tianruo Yang, and Yan Zhang, editors, *Autonomic Computing and Networking*, pages 3–30. Springer, 2009.
18. R. Calinescu and M. Kwiatkowska. Using quantitative analysis to implement autonomic it systems. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 100–110, Washington, DC, USA, 2009. IEEE Computer Society.
19. G. Canfora, M. Di Penta, R. Esposito, and M.L. Villani. A framework for QoS-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, 2008.
20. V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola. QoS-driven runtime adaptation of service oriented architectures. In *ESEC/FSE 2009, Proceedings*, pages 131–140. ACM, 2009.
21. G. Casale, R. Muntz, and G. Serazzi. Geometric bounds: A noniterative analysis technique for closed queueing networks. *IEEE Trans. Comput.*, 57(6):780–794, 2008.
22. 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.
23. B.H.C. Cheng, R. de Lemos, G. 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.
24. A. Clark, S. Gilmore, J. Hillston, and M. Tribastone. Stochastic process algebras. In *7th Intern. School on Formal Methods, SFM*, volume 4486 of *LNCS*, pages 132–179. Springer, 2007.
25. Ardagna D. and Mirandola R. Per-flow optimal service selection for web services based processes. *Journal of Systems and Software*, 83(8):1512–1523, 2010.
26. E. Di Nitto, C. Ghezzi, A. Metzger, M.P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.
27. M.B. Dwyer, J.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In *Proc. 21th International Conference on Software Engineering (ICSE99)*, pages 411–420. ACM Press, 1999.

28. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. 31st International Conference on Software Engineering (ICSE09)*, pages 111–121, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
29. A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *33 International Conference on Software Engineering (ICSE11)*, accepted for publication.
30. Chaffe G., P. Doshi, J. Harney, S. Mittal, and B. Srivastava. Improved adaptation of web service compositions using value of changed information. In *ICWS*, pages 784–791. IEEE Computer Society, 2007.
31. S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In *QoSA, Quality of Software Architecture*, Lecture Notes in Computer Science. Springer, 2008.
32. S-W. Garlan, D. and Cheng, A-C Huang, B.R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.
33. C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services*, pages 237–264. Springer, 2007.
34. C. Ghezzi, A. Motta, V. Panzica La Manna, and G. Tamburrelli. Qos driven dynamic binding in-the-many. In *QoSA*, pages 68–83, 2010.
35. Carlo Ghezzi and Giordano Tamburrelli. Reasoning on non-functional requirements for integrated services. In *RE '09: Proceedings of the 17th International Conference on Requirements Engineering*, Atlanta, USA, 2009.
36. V. Grassi. Architecture-based reliability prediction for service-oriented computing. In *Workshop on Architecting Dependable Systems, WADS*, volume 3549 of *LNCS*, pages 279–299. Springer, 2004.
37. V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci*, 153(2):117–133, 2006.
38. L. Grunske. Specification patterns for probabilistic quality properties. In Robby, editor, *30th International Conference on Software Engineering (ICSE 2008)*, pages 31–40. ACM, 2008.
39. H. Guo, J. Huai, H. Li, T. Deng, Y. Li, and Z. Du. ANGEL: Optimal Configuration for High Available Service Composition. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 280–287. IEEE Computer Society, 2007.
40. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
41. J. Harney and P. Doshi. Speeding up adaptation of web service compositions using expiration times. In *World Wide Web (WWW)*, pages 1023–1032. ACM, 2007.
42. 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.
43. A. Immonen and E. Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and System Modeling*, 7(1):49–65, 2008.
44. R. Jain. *The Art of Computer Systems Performance Analysis—Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, 1991.
45. J-P. Katoen, T. Kemna, I.S. Zapreev, and D.N. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In Orna Grumberg and Michael

- Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2007, Proceedings*, volume 4424 of *LNCS*, pages 87–101. Springer, 2007.
46. J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *QEST*, pages 243–244, Los Alamos, CA, USA, 2005. IEEE Computer Society.
 47. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
 48. T. Kerola. The composite bound method for computing throughput bounds in multiple class environments. *Performance Evaluation*, 6(1):1–9, 1986.
 49. S. Konrad and B.H.C. Cheng. Real-time specification patterns. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 05)*, pages 372–381. ACM Press, 2005.
 50. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16:1293–1306, November 1990.
 51. M. Kwiatkowska. Quantitative verification: Models, techniques and tools. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 449–458. ACM Press, September 2007.
 52. M.Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *Int. Journal on Software Tools for Technology Transfer(STTT)*, 6(2):128–142, August 2004.
 53. M.Z. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Proceedings*, volume 4144 of *LNCS*, pages 234–248. Springer, 2006.
 54. M.Z. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29(1):33–78, 2006.
 55. E. D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.
 56. Caporuscio M., Funaro M., and Ghezzi C. Architectural issues of adaptive pervasive systems. In *Graph Transformations and Model-Driven Engineering*, pages 492–511, 2010.
 57. Marzolla M. and Mirandola R. Performance aware reconfiguration of software systems. In *Computer Performance Engineering - 7th European Performance Engineering Workshop, EPEW 2010, Bertinoro, Italy, September 23-24, 2010. Proceedings*, volume 6342 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2010.
 58. S. Maoz. Using model-based traces as runtime models. *IEEE Computer*, 42(10):28–36, 2009.
 59. M.A. Marsan. Stochastic petri nets: an elementary introduction. In *Advances in Petri Nets*, pages 1–29, Berlin - Heidelberg - New York, June 1989. Springer.
 60. A. Martens, H. Koziolok, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proc. first joint WOSP/SIPEW international conference on Performance engineering*, pages 105–116, New York, NY, USA, 2010. ACM.
 61. D.A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa. A framework for utility-based service oriented design in sassy. In *Proc. first joint WOSP/SIPEW int. conf. on Performance engineering*, pages 27–36, New York, NY, USA, 2010. ACM.

62. B. Morin, O. Barais, J-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *IEEE Computer*, 42(10):44–51, 2009.
63. Martin L. Puterman. *Markov Decision Processes*. Wiley, 1994.
64. F. Raimondi, J. Skene, and W. Emmerich. Efficient online monitoring of web-service slas. In *SIGSOFT FSE*, pages 170–180. ACM, 2008.
65. S.M. Ross. *Stochastic Processes*. Wiley New York, 1996.
66. M. Salehie, S. Li, R. Asadollahi, and L. Tahvildari. Change support in adaptive software: A case study for fine-grained adaptation. In *EASE '09: Proc. Sixth IEEE Conf. and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 35–44, Washington, DC, USA, 2009. IEEE Computer Society.
67. N. Sato and K.S. Trivedi. Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2007.
68. K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 266–280. Springer Berlin Heidelberg, 2005.
69. R.N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *WICSA/ECSA*, pages 171–180. IEEE, 2009.
70. 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.
71. L. Wang, N.J. Dingle, and W.J. Knottenbelt. Natural language specification of performance trees. In Nigel Thomas and Carlos Juiz, editors, *Proceedings of the 5th European Performance Engineering Workshop, EPEW 2008*, volume 5261 of *LNCS*, pages 141–151, 2008.
72. L. Zeng, B. Benatallah, A.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Trans. Software Eng.*, 30(5):311–327, 2004.
73. T. Zheng, M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. 34(3):391–406, 2008.