Self-adaptive software needs quantitative verication at runtime

Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, Raffaela Mirandola

Self-adaptive software needs quantitative verification at runtime*

Radu Calinescu
Department of Computer Science
University of York
Deramore Lane
York YO10 5GH, UK
radu.calinescu@york.ac.uk

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

Marta Kwiatkowska
Department of Computer Science
University of Oxford
Wolfson Building, Parks Road
Oxford OX1 3QD, UK
Marta.Kwiatkowska@cs.ox.ac.uk

Raffaela Mirandola
Politecnico di Milano
DEI–DeepSE Group
Piazza Leonardo da Vinci, 32
Milano, Italy
mirandola@elet.polimi.it

1 Introduction

Software is surreptitiously becoming the backbone of modern society. Most human activities are either software enabled or entirely managed by software. Examples range from healthcare and transportation to commerce and manufacturing. In all these applications, one requirement is becoming common: software must adapt continuously, to respond to changes in application objectives and in the environment in which it is embedded. The autonomic computing vision of systems that respond to change by evolving in a self-managed manner while running and providing service [4, 9, 20] is rapidly becoming reality.

A second key requirement that today's software is increasingly expected to fulfil is dependability. As software is used more and more in business-critical and safety-critical applications, the adverse impact of unreliable or unpredictable software is growing. Damaging effects varying from loss of business to loss of human life are no longer uncommon.

These two requirements of modern software—adaptiveness and dependability—have typically been the concern of different research communities. Researchers from the area of autonomic computing have been developing adaptive software systems successfully for the past decade

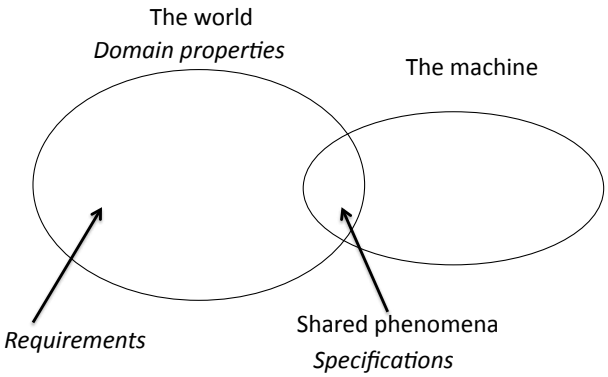


Figure 1: The world and the machine.

[19, 30]. In contrast, several mathematically-based modelling and analysis techniques are traditionally practised in off-line manner, to improve the dependability, performance and operating cost (e.g., energy consumption) of software. These techniques include model checking [10] and, more recently, a mathematically-based technique for establishing the correctness, performance and reliability of systems that exhibit stochastic behaviour termed *quantitative verification* [21]. These *formal verification techniques* prevent errors from reaching the final implementation that is delivered for use, or at least remove them when a new version of the software is deployed.

We argue that the only way to achieve the dependable software adaptation required by an increasing number of applications is to bring the techniques developed by these two research communities together. Consequently, quantitative verification and model checking must also be used at runtime, to predict and identify requirement violations, to plan the adaptation steps necessary to prevent or recover from such violations, and to obtain irrefutable proofs that the reconfigured software will comply with its requirements. Software tools that implement flexible and low-overhead variants of both techniques have to be run entirely automatically, to support all these stages of the adaptation process. The result of adopting this approach is software that supports not only automated changes, but also their continual formal verification, to guarantee that the software continues to meet its requirements as it evolves.

We use our exploration of this new paradigm in a range of projects to explain how quantitative verification can extend its operation to runtime, and illustrate what can be achieved through it. A range of complementary approaches to using formal verification techniques in runtime scenarios are then summarised and used to place our results in context. Looking ahead, we present the main research challenges that must be addressed to make the transition of formal verification to runtime efficient and effective.

2 A Reference Framework

Software *evolution* has been recognized as a key distinctive feature since the early 1970s by many researchers, and most notably by Belady and Lehman [24]. Indeed, evolution is perhaps the most important aspect that distinguishes software from other artifacts produced by humans.

To shed light on software evolution, we refer to Jackson and Zave's seminal work on requirements [31]. In this work, a clear distinction is made between the *world* and the *machine*. As illustrated in Figure 1, the machine is the system to be developed via software; the world (i.e., the environment) is the portion of the real-world that needs to be affected by the machine. The ultimate purpose of building a machine is always to be found in the world. *Requirements* are thus statements on the desired phenomena occurring in the world. They should not refer to phenomena occurring inside the machine, which concern only the implementation. 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 prescriptive statement of the relations on shared phenomena that must be enforced by the system to be developed.

To develop a machine, software engineers must first derive a specification from the requirements. To do so, they need to understand the relevant assumptions that need to be made about the environment in which the machine is expected to work, namely the assumptions that affect the achievement of the desired results. These environment assumptions are often called *domain knowledge*. Quoting from [31]: *"The primary role of domain knowledge is to bridge the gap between requirements and specifications"*.

The set of relevant assumptions captured by domain knowledge enables us to prove that—through the machine—we achieve the desired requirements. Let R and S be (prescriptive) statements that describe the requirements and the specification in some formal notation, respectively, and let D be the (descriptive) formal statements that specify the domain assumptions. If S and D are all satisfied and consistent, then software engineers should be able to prove that R also holds. Formally:

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

which states that S ensures satisfaction of the requirements R in the context of the domain properties D. Figure 2 shows how this formalism applies to a simplified version of a medical assistance system from [5].

Domain assumptions play a fundamental role in building systems that satisfy the requirements. Engineers need to know upfront how the environment in which their software will be embedded works, since the software can achieve the expected goals only under certain assumptions on the behavior of the domain described by D. Should these assumptions be invalidated, the software developed will most likely fail to satisfy the requirements.

Software evolution deals with changes that affect the machine, i.e., changes to the specification S, which then cause changes also in the implementation. Software evolution is triggered by a violation of the correctness criterion in formula (1), which is discovered after the software is released. This violation may occur because:

(i) The implemented machine does not satisfy the specification.

(ii) The actual behavior of the environment diverges from the domain assumptions D made when the specification was devised.

(iii) The requirements R do not capture the actual goals we wish to achieve in the real world.

Traditionally, a response to these changes is handled by modifying the software off-line, during a *maintenance* phase. Case (i) above corresponds to *corrective maintenance*. Case (ii)

corresponds to *adaptive maintenance*. That is, S needs to be changed to achieve satisfaction of the requirements under the newly discovered relevant domain properties. Case (iii) corresponds to *perfective maintenance*. That is, changes in R require that S also changes. For example, business goals might evolve over time or new features might be requested by users of the application. Because maintenance is an off-line activity, software returns to the development stage, where the necessary changes are analyzed, prioritized, and scheduled. Changes are then handled by modifying the specification, design, and implementation of the application. The evolved system is then verified, typically via some kind of regression testing, and redeployed.

Off-line maintenance does not meet the needs of emerging application scenarios in which systems need to be continuously running and capable of adapting autonomously (and as soon as the need for change is detected) while they operate. In this paper, we mostly focus on changes that occur in the environment (D). We use the term *self-adaptive software* to indicate that the software has autonomous capabilities through which it tries to satisfy criterion (1) as changes to D, which lead to violations of type (ii), are detected. These changes are typically due to:

- high uncertainty about the behaviour of the environment when the application is developed, and
- high variability in the behaviour of the environment as the application is running.

This paper focuses mostly on system properties that can be expressed quantitatively and require quantitative verification, such as reliability, performance and energy consumption. Increasingly, the requirements that software must guarantee are expressed in terms of these properties. These properties are heavily influenced by the way the environment behaves, and thus environment assumptions are crucial to engineering software systems that satisfy such requirements. For example, assumptions on the user behavior profiles may affect performance.

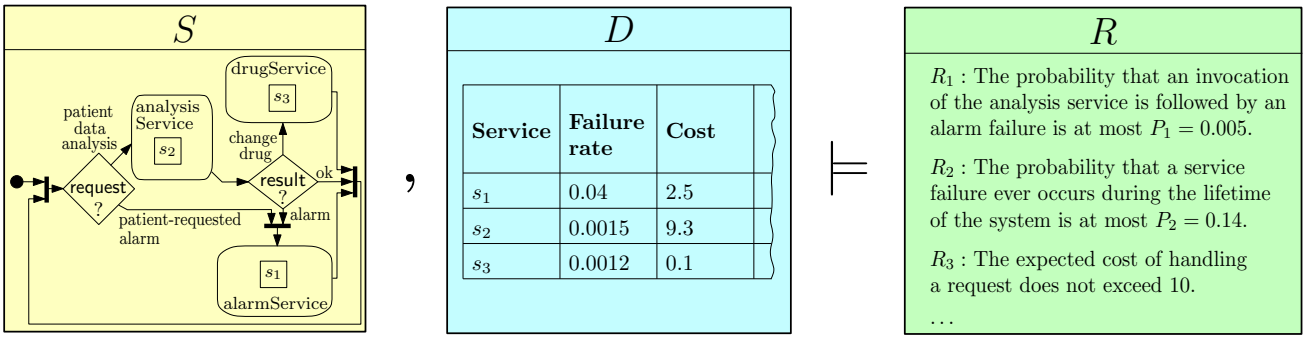


Figure 2: A medical assistance (MA) application whose specification S, domain assumptions D and requirements R satisfy eq. (1). The specification S describes a service-based system for the MA application, comprising the possibility to perform an analysis of the patient data (provided by service s_2) or to send a patient-requested alarm (service s_1). In the former case, the result of the analysis determines whether the system should change the drugs prescribed to the patient (service s_3), send an alarm (service s_1) or do nothing. D describes the domain assumptions in terms of failure rates and costs of the available concrete services s_1, s_2 and s_3. The requirements R for the MA application comprise reliability-related requirements stating, for example, the maximum tolerated probability of a service failure.

Self-adaptation can also be explained with reference to *autonomic computing* [20], which employs a monitor-analyse-plan-execute (MAPE) closed control loop to achieve self-management within computer systems. In this reference model, core to the operation of the component that realises the adaptation process (termed an *autonomic manager*) is the *knowledge* combining the assumptions D and specification S. This knowledge is updated continually through monitoring the environment and the system through *sensors*, and is used to analyse whether the user-specified requirements R continue to be satisfied. Whenever this is no longer the case, appropriate system changes are planned and "executed" through *effectors*.

We argue that formal verification techniques such as quantitative verification and model checking can provide the foundational support required by the multiple stages of the MAPE autonomic computing loop, in order to integrate the flexibility achieved through adaptation with the dependability that is needed for today's pervasive software. This is exemplified in the next section for quantitative verification, and summarised in Section 4 for a range of related software modelling, specification and analysis techniques.

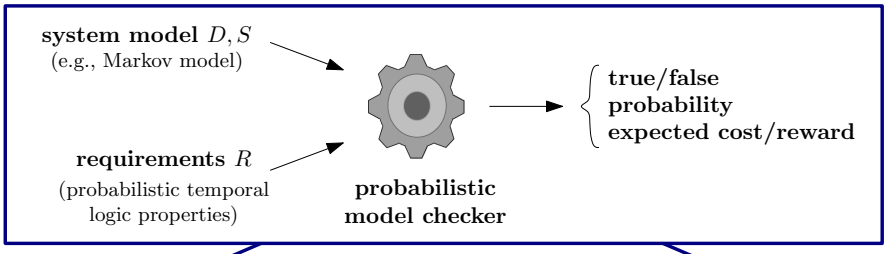
3 Quantitative Verification at Runtime

Quantitative verification is a mathematically-based technique for establishing the correctness, performance and reliability of systems that exhibit stochastic behaviour [21]. Given a finite mathematical model of a software system and requirements specified formally in temporal logics extended with probabilities and costs/rewards, an exhaustive analysis of the model's compliance with these requirements is performed (Figure 3). Example requirements that can be established using the technique include the probability that a fault occurs within a specified time period, and the expected response time of a software system under a given workload.

Using *quantitative verification at runtime* can support several stages of the software adaptation process:

1. In the *monitoring* stage, it supports the precise and rigorous modelling of the domain assumptions D (Figure 4). This involves augmenting the software system with a component responsible for the continuous updating of the parameters of a quantitative model of the system based on observations of the system behaviour. For instance, for the discrete-time Markov chain in Figure 4, this component can update the service failure rates x, y and z in line with the observed service behaviour by using the Bayesian learning methods we introduced in [6, 12]. Likewise, the parameters of the continuous-time Markov chain models typically used to model performance-related aspects of software systems can be updated using Kalman filter estimators [32].

2. In the *analysis* stage, a quantitative verification tool can be invoked automatically to detect (and sometimes to predict) requirement violations. To achieve this, the tool verifies the formally specified requirements R (Figure 4) against the quantitative model obtained through combining the specification S with the updated domain assumptions D from the monitoring stage. When a requirement $r \in R$ is no longer satisfied by the updated model, we distinguish between two different scenarios. In the first scenario, the observation that triggered the model update was caused by observing system operations related to r, so the violation of requirement r is *detected*. In the second scenario, the updated model that does not satisfy r was obtained by observing system operations unrelated to this requirement, so



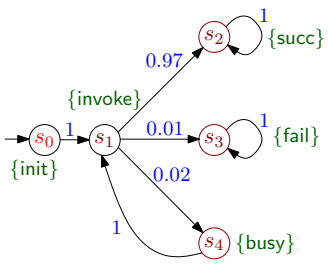
System model D, S: discrete-time Markov chain (DTMC)

Definition 1. A DTMC over an atomic proposition set AP is a four-element tuple

$$M = (\mathcal{S}, s_{init}, \mathbf{P}, L),$$

where $\mathcal{S}$ is a finite set of states, $s_{init} \in \mathcal{S}$ is the initial state, $\mathbf{P} : \mathcal{S} \times \mathcal{S} \to [0,1]$ is the transition probability matrix, and $L : \mathcal{S} \to 2^{AP}$ is a labelling function. For all $s, s' \in \mathcal{S}$, $\mathbf{P}(s, s')$ represents the probability that the system transitions to state s' when it is in state s; and $\sum_{s' \in \mathcal{S}} \mathbf{P}(s, s') = 1$ for all $s \in \mathcal{S}$.

Example 1. The DTMC below models the behaviour of a service used in a service-based system. The service invocation succeeds with probability 0.97, fails with probability 0.01, and needs to be retried (because the service is busy) with probability 0.02.



$M = (\mathcal{S},\ s_{init},\ \mathbf{P},\ L)$

where:

$\mathcal{S} = \{s_0, s_1, s_2, s_3, s_4\}$

$s_{init} = s_0$

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.97 & 0.01 & 0.02 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$AP = \{\text{init, invoke, succ, fail, busy}\}$

$L = \{s_0 \mapsto \{\text{init}\}, s_1 \mapsto \{\text{invoke}\}, s_2 \mapsto \{\text{succ}\}, s_3 \mapsto \{\text{fail}\}, s_4 \mapsto \{\text{busy}\}\}$

Requirements R: Probabilistic computation tree logic (PCTL)

Definition 2. The PCTL formulas over a set of atomic propositions AP are defined inductively by:

$$\phi ::= \mathit{true} \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathrm{P}_{\bowtie p}[\mathrm{X}\phi] \mid \mathrm{P}_{\bowtie p}[\phi \mathrm{U}^{\leq k} \phi] \mid \mathrm{P}_{\bowtie p}[\phi \mathrm{U} \phi]$$

where $a \in AP$, $p \in [0,1]$ is a probability, $\bowtie \in \{<, >, \leq, \geq\}$ is a relational operator, and $k \in N$.

Definition 3 (PCTL semantics for DTMCs). PCTL formulas are interpreted over states of a DTMC $M = (\mathcal{S}, s_{init}, \mathbf{P}, L)$: given a state $s \in \mathcal{S}$ and a PCTL formula ϕ, $s \models \phi$ means "ϕ is satisfied in state s" or "ϕ is true in state s". We have:

- $s \models \mathit{true}$ always holds, $s \models a$ holds iff $a \in L(s)$, $s \models \neg\phi$ iff $s \models \phi$ is false, and $s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$;
- $s \models \mathrm{P}_{\bowtie p}[\mathrm{X}\phi]$ iff, across all *paths* (i.e., sequences of states with non-null probability of transition between any successive states) $s_{i_1}, s_{i_2}, \ldots$ with $s_{i_1} = s$, the probability that $s_{i_2} \models \phi$ satisfies $\bowtie p$;
- $s \models \mathrm{P}_{\bowtie p}[\phi_1 \mathrm{U}^{\leq k} \phi_2]$ iff, across all paths $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ with $s_{i_1} = s$, the probability that there is a $j \leq k$ such that $s_{i_j} \models \phi_2$ and, for every $l < j$, $s_{i_l} \models \phi_1$ satisfies $\bowtie p$;
- $s \models \mathrm{P}_{\bowtie p}[\phi_1 \mathrm{U} \phi_2]$ iff, across all paths $s_{i_1}, s_{i_2}, \ldots$ with $s_{i_1} = s$, the probability that there is a $j \geq 0$ such that $s_{i_j} \models \phi_2$ and, for every $l < j$, $s_{i_l} \models \phi_1$ satisfies $\bowtie p$.

Example 2. The requirement that the invocation of the service from Example 1 succeeds with probability of at least 0.985 is expressed in PCTL as '$\mathrm{P}_{\geq 0.985}$[true U succ]'.

Figure 3: Quantitative verification of reliability requirements using discrete-time Markov chains to express the specification S and domain assumptions D, and probabilistic computation tree logic to formalise the requirements R. Quantitative verification of performance requirements can be carried out using complementary formalisms such as continuous-time Markov chains (CTMCs) and continuous stochastic logic (CSL), and cost-related requirements can be verified using variants of these formalisms augmented with costs/rewards [21].

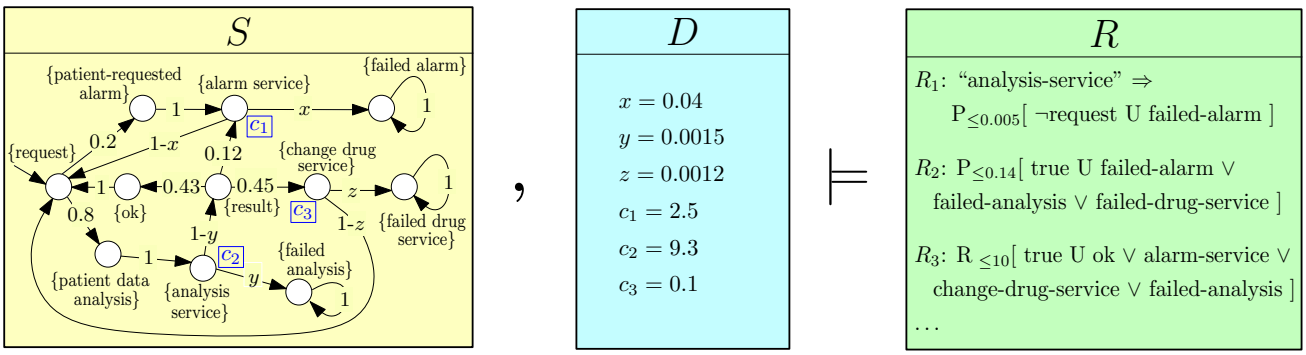


Figure 4: Formalisation of $S, D \models R$ for the software system in Figure 2. To support quantitative verification, the specification S is modelled as a *discrete-time Markov chain*, comprising states for all possible system configurations (represented as circles) and state transitions (depicted as edges annotated with the probabilities of the associated transitions). The domain assumptions D comprise variables that parameterise this model, reflecting the fact that service failure rates and costs may vary in a real-world system. Finally, the requirements R are expressed in *probabilistic computation tree logic* extended with the rewards operator R, for models annotated with costs.

the violation of requirement r is *predicted*. For example, an observed failure of the alarm service from the system in Figure 2 may yield an updated model that ceases to satisfy requirement R_1 from Figures 2 and 4. The alarm service invocation that failed could have been initiated by two events: (a) an abnormal result from the analysis service—in which case the analysis detects the violation of R_1; or (b) a patient request—in which case the violation of R_1 is predicted.

3. The *planning* stage is carried out when the analysis stage finds that requirements such as response time, availability and cost are or will be violated. As explained in Section 2, adaptive maintenance leading to appropriate updates of the specification S is necessary in these circumstances. Quantitative verification can support planning by suggesting adaptive maintenance steps whose execution ensures that the system continues to satisfy its requirements despite the changes identified in the monitoring phase. As an example, suppose that the medical assistance system from Figure 2 can select its alarm and analysis services dynamically, from sets of services provided by multiple third parties. Although functionally equivalent, these services will typically be characterised by different levels of reliability, performance and cost. A quantitative verification tool invoked automatically at runtime supports this dynamic service selection through establishing which combinations of alarm and analysis services (i.e., which specifications S) satisfy the requirements R at each time instant.

We used the probabilistic model checker PRISM [18] to validate the approach described above in domains ranging from dynamic power management [7] and data-centre resource allocation [8] to quality-of-service optimisation in service-based systems [5, 12]. The last of these applications was used as a basis for the simplified example of a software system from Figures 2 and 4, and is presented in more detail in Figure 5. Success in these projects indicates that employing quantitative verification in runtime scenarios can augment software systems with self-adaptation capabilities *in predictable ways*.

Service-based systems (SBSs) are software applications built through the composition of loosely-coupled services from different providers. SBSs are used in application domains that include e-commerce, on-line banking and healthcare, and are required to operate in scenarios that involve frequent changes in environment and requirements. As a result, the effectiveness of SBSs is increasingly dependent on their ability to self-adapt. One way of devising *self-adaptive SBSs* is to dynamically select the services that implement their operations from sets of functionally equivalent services that are associated with different levels of performance, reliability and cost.

The diagram below depicts a simplified version of a self-adaptive medical assistance SBS from [2]. The upper-left corner of the diagram shows the SBS specification S, domain assumptions D and requirements R at the initial time instant t_1, when the requirements are satisfied: $S, D \models R$. However, as the failure rate of the alarm service used by the SBS (i.e., s_1^2) is observed to increase through Bayesian learning in the monitoring stage of the MAPE autonomic computing loop, the runtime use of quantitative verification in the analysis stage establishes that the requirements are violated at time instant t_2: $S, D' \not\models R$. To remedy this, the planning step of the MAPE loop uses quantitative verification to select another service for the alarm operation. Accordingly, a new specification S' is employed to ensure that the requirements are again satisfied at time instant t_3: $S', D' \models R$.

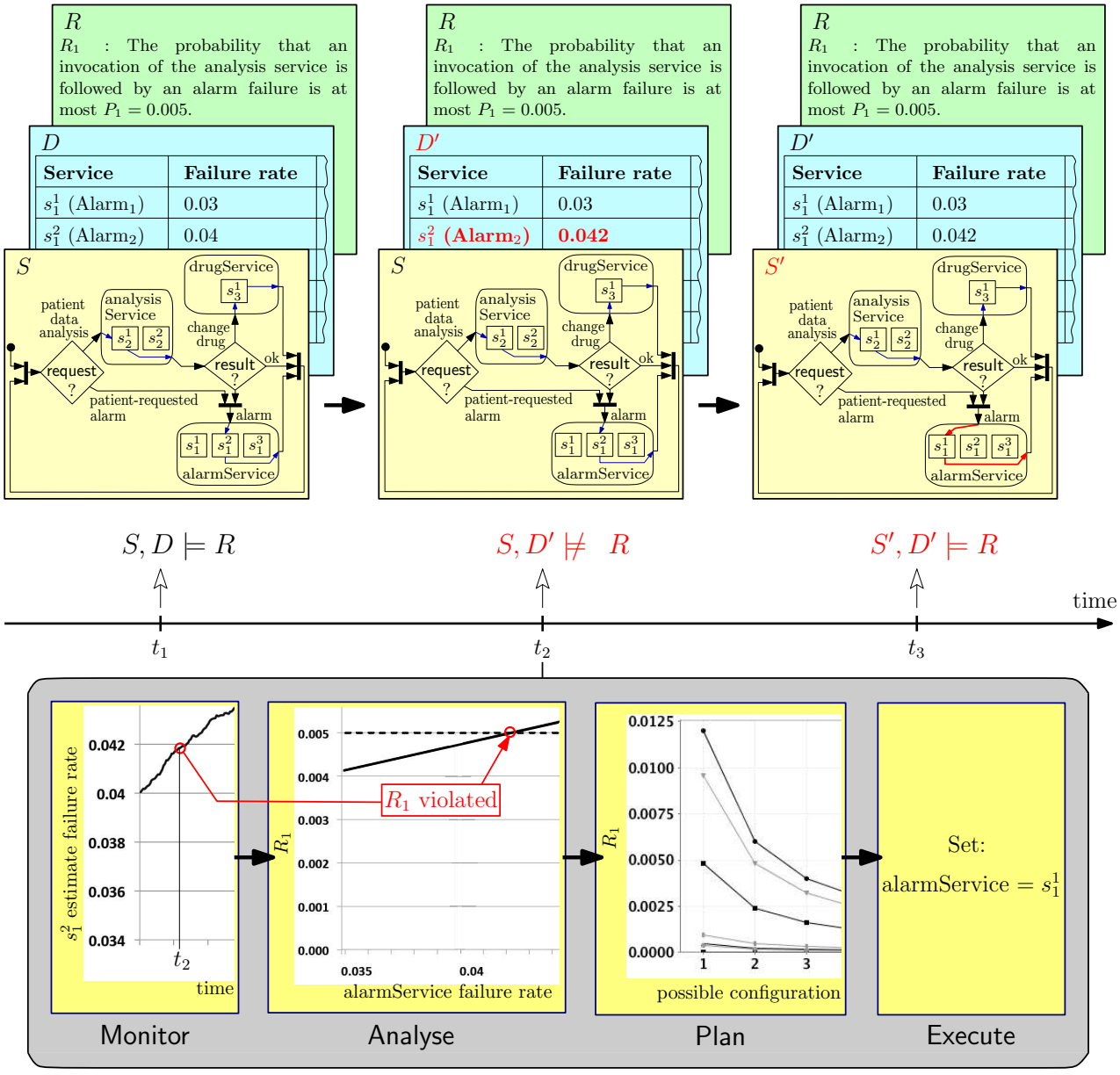


Figure 5: Self-adaptive service-based systems using quantitative verification at runtime

Using Markovian models at a carefully chosen level of abstraction enabled the adaptive systems described above to operate with acceptable overheads[1] for medium and small-sized systems. Scaling beyond these system sizes requires faster runtime verification algorithms, and our recent research to devise such algorithms shows great promise [13, 22]. This research exploits a unique characteristic of the approach, namely the fact that the system model and the verified requirements are unlikely to change significantly between adaptation steps.

The work in [13] pre-computes the probabilities associated with reliability-related requirements of a software system as symbolic expressions parameterised by the domain assumptions. As a simple example, the "probability that an invocation of the analysis service is followed by an alarm failure" associated with requirement R_1 for the system in Figure 4 can be pre-computed as $P_1 = (1 - y) \times 0.12 \times x$, where the parameters x and y represent the failure rates of the alarm service and the analysis service, respectively. This "once only" *pre-computation step* is complemented by a *runtime verification step* in which the symbolic expressions are evaluated for the actual values of the system parameters. In our example, the runtime verification step consists of calculating the new value of P_1 each time the domain assumptions about the parameters x or y change as a result of the runtime monitoring. The overheads associated with the pre-computation step are comparable to those of standard quantitative verification, but the overhead to evaluate a set of symbolic expressions in the runtime verification step is negligible irrespective of the system size.

The approach in [22] achieves similar improvements by using an *incremental* technique for the verification of Markov decision processes, which subsume discrete-time Markov chains discussed previously, for the case where the probability value can vary at runtime. This technique exploits the fact that small changes in the model being verified often affect only a small subset of its strongly connected components (SCCs). By reusing the verification results associated with the SCCs unaffected by change between one adaptation step and the next, the technique substantially reduces the computation cost of re-verification of the requirement. A symbolic implementation of the technique was shown to reduce the verification time by up to two orders of magnitude [22].

The scalable verification techniques described above enable the use of quantitative verification at runtime to develop larger adaptive software solutions than previously possible.

4 Related Work

In recent years, several research communities have made advances towards the integration of formal verification techniques into the runtime software adaptation process. These results are contrasted with our work on quantitative verification at runtime, as described below.

Crow and Rushby's work on a theory of fault detection, identification and reconfiguration [11] represents an early precursor of using formal verification in the runtime software adaptation process. In his later work on *runtime certification* [29], Rushby emphasises the need for runtime configuration, and argues that any software reconfiguration at runtime must be accompanied by a certification of its dependability. In this context, Rushby argues that the use of formal verification represents an effective approach to achieving runtime certification, and describes a framework that enables it—including through the runtime use of "methods related to model checking" [29].

[1] Typical self-adaptation times ranged from hundreds of milliseconds for the disk-drive power management solution in [8] to a few seconds for the applications in [5, 7, 12].

The range of (qualitative) correctness properties whose realisation is supported by this framework (e.g., safety and reachability) complement the spectrum of reliability- and performance-related properties that can be managed using our quantitative verification at runtime approach.

Recent advances in the area of *models @ runtime* provide additional evidence that the runtime use of models has the ability to support software adaptation. Morin *et al.* [26] describe a method for developing adaptive software by predefining a set of system configurations, and using *aspect-oriented model reasoning* to select the most suitable of these configurations at runtime. Different configurations may be associated with different quality-of-service properties, or with different sets of supported services. The approach is likened to a "dynamic software product line" [26]. Similar results have been obtained through the use of architectural models as a guide for the software adaptation process [14, 15]. These approaches employ general and user-defined constraint verification techniques to change the architecture of a software system at a coarse level (e.g, by switching between two versions of a user interface). In contrast, the runtime use of quantitative verification also supports fine-grained adaptation of system parameters, e.g., by continually adjusting the amount of CPU allocated to the services of a software system [5].

The *runtime verification* community proposes that program execution traces obtained through monitoring are analysed at runtime, to establish in real time whether the software satisfies or violates given correctness requirements. These corectness requirements are expressed using formalisms that include temporal logics [25, 27], state machines [2], regular expressions [1], rule systems [3] and action-based contract languages [23]. Our approach is different and complementary, since it focuses on quantitative verification and continuous monitoring of environment phenomena. Another related research area is *dynamic software composition*, which provides approaches (for example, based on AI planning techniques [28]) that can support adaptive reactions triggered by requirements violations.

5 Research Opportunities

Adaptive software systems are increasingly built in practice. There has been much activity in the area in recent years, and a number of contributions that go beyond ad-hoc practices have already been produced. Despite these advances, much remains to be done to support the development of *predictable adaptive software* via a formal, systematic and disciplined approach. Hereafter we elaborate on the main research areas in which significant work is required to better integrate formal verification techniques into software adaptation. The list is not exhaustive, but reflects the key challenges encountered and foreseen in our work and that of the research communities mentioned in the previous section.

A first area of research is *discovery* and *model learning*. We expect future software systems to operate in environments populated by active devices and appliances that will offer services. These environments will be highly dynamic. For example, the context may change due to movement in space. As another example, new services may be deployed and dynamically discovered. These services (and the components providing them) may not know each other, but they might still try to understand what each can do and possibly cooperate to achieve common goals. How can this be realised? How can a component learn what another component offers, given different levels of visibility into the internals of components? For example, how far can we go in the case of *black-box visibility*, when only observations of the external behavior of a component are available? Our preliminary work in this area aims to infer the functional behavior of a (stateful)

component from observations of inputs and outputs at the level of its API [16]. Our method applies suitable learning strategies, and is largely based on an assumption of *regularity* in the behaviour of components. It has been tested successfully in the case of Java data abstractions [17], but more needs to be done to make the approach general and practical.

A second area of research concerns the effective integration of *formal verification* and *self-adaptation*. The goal is to develop a repertoire of techniques that can provide a timely and effective reaction to detected violations of the requirements. The strategies to follow are in fact very much domain and application dependent. For example, the techniques for speeding up runtime quantitative verification presented in Section 3 are justified whenever the time taken by the traditional variant of the technique is incompatible with the time needed for reaction. A catalogue of possible reaction strategies should in turn be available at runtime. An interesting approach would be to handle adaptation within the model-driven framework. Since models are "kept alive" at runtime, once the need for adaptive reactions is identified, it would be useful to perform self-adaptation at the model level, and then to re-play model-driven development to derive an implementation through a chain of automatic transformations. If changes can be anticipated at design time, they may be reified as variation points in the models, and variations would then be generated dynamically to achieve adaptation. In the more difficult case of unanticipated changes, it might still be possible to devise a number of possible adaptation strategies and tactics that can be attempted at runtime.

Yet another key research area involves addressing the problems that arise from new execution platforms, such as *cloud computing*. So far we assumed that changes originate either in the requirements or in the domain assumptions. But with the advent of cloud computing, the infrastructure on which our *machine* works may also change. To exploit the full potential of the *service* paradigm, we must complement the traditional service-oriented architecture view of *software-as-a-service* with a perception of the platform and infrastructure on which the software is run as services too. The use of a single abstraction to reason about both the machine and the infrastructure may pave the ground to "holistic" solutions. Self-adaption cannot be seen only at application level, but we must deploy probes, conceive analysis techniques, and identify solutions able to drive self-adaption of the system as a whole. Adaptations at application level must consider the implications on the lower levels; conversely, these levels should provide the means enabling the application to execute effectively. Adaptation becomes much more of an inter-level problem than a set of isolated intra-level solutions.

Moreover, the adoption of cloud infrastructures will also impose a shift from client-side "proprietary" computing resources to "shared" ones. Web services made us think of the distributed ownership of our applications; the cloud will make us think of the distributed ownership of our infrastructure. To some extent, this is already the case when we build software applications using services run and shared by others. Yet, clouds will exacerbate the problem significantly. The execution of one application will compete with the execution of another, turning self-verification and self-adaption into infrastructure-wide requirements.

6 Conclusion

We discussed the potential and challenges associated with the runtime use of techniques such as quantitative verification and model checking as a way to obtain dependable self-adaptive software. Our experience with using quantitative verification at runtime on a range of projects

shows that this technique can support software adaptation in several ways. First, it can be used to identify and, sometimes, to predict requirement violations. Second, it supports the rigorous planning of the reconfiguration steps that self-adaptive software employs to recover from such requirement violations. Last but not least, it can provide irrefutable proofs that the selected reconfiguration steps are correct. The result is software that supports not only automated changes, but also their continual formal analysis, to verify that the software is guaranteed to meet its requirements as it evolves.

Acknowledgments

This research has been partially funded by the European Commission Programme IDEAS-ERC, Project 227977-SMScom, by the UK Engineering and Physical Sciences Research Council Grants EP/F001096/1 and EP/H042644/1, by the European Commission FP 7 project CONNECT (IST 231167), and by the ERC Advanced Grant VERIWARE.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 345–364, New York, NY, USA, 2005. ACM.

[2] Howard Barringer and Klaus Havelund. A Scala DSL for trace analysis. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 57–72. Springer Berlin / Heidelberg, 2011.

[3] Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce. Rule systems for runtime verification: A short tutorial. In Saddek Bensalem and Doron Peled, editors, *Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin / Heidelberg, 2009.

[4] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M. Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70, 2009.

[5] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37:387–409, 2011.

[6] R. Calinescu, K. Johnson, and Y. Rafiq. Using observation ageing to improve Markovian model learning in QoS engineering. In *Proceedings 2nd ACM/SPEC International Conference on Performance Engineering*, pages 505–510, 2011.

[7] R. Calinescu and M. Kwiatkowska. CADS*: Computer-aided development of self-* systems. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*

(FASE 2009), volume 5503 of *Lecture Notes in Computer Science*, pages 421–424. Springer, 2009.

[8] R. Calinescu and M. Kwiatkowska. Using quantitative analysis to implement autonomic IT systems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 100–110, 2009.

[9] Betty H. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer-Verlag, 2009.

[10] E. M. Clarke and F. Lerda. Model checking: Software and beyond. *Journal of Universal Computer Science*, 13(5):639–649, 2007.

[11] J. Crow and J. Rushby. Model-based reconfiguration: Diagnosis and recovery. NASA Contractor Report 4596, NASA Langley Research Center, Hampton, VA, May 1994. (Work performed by SRI International).

[12] 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.

[13] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 341–350. IEEE Computer Society, 2011.

[14] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Softw.*, 23:62–70, March 2006.

[15] D. Garlan and B. R. Schmerl. Using architectural models at runtime: Research challenges. In *EWSA*, pages 200–205, 2004.

[16] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *Proceedings of the 31st International Conference on Software Engineering*, pages 430–440. IEEE Computer Society, 2009.

[17] C. Ghezzi, A. Mocci, and G. Salvaneschi. Automatic cross validation of multiple specifications: A case study. In *Proceedings of Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2010.

[18] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[19] M.C. Huebscher and J.A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.

[20] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, 2003.

[21] M. Kwiatkowska. Quantitative verification: Models, techniques and tools. In *Proc. 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. Foundations of Software Engineering*, pages 449–458. ACM Press, September 2007.

[22] M. Kwiatkowska, D. Parker, and H. Qu. Incremental quantitative verification for Markov decision processes. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks*, 2011.

[23] M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *Proceedings 6th International Symposium on Automated Technology forVerification and Analysis (ATVA'08)*, 2008.

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

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

[26] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. *Computer*, 42:44–51, October 2009.

[27] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In *Proceedings 14th International Symposium on Formal Methods (FM'06)*, pages 573–586, 2006.

[28] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin / Heidelberg, 2005.

[29] J. M. Rushby. Runtime certification. In *Proceedings 8th International Workshop on Runtime Verification (RV'08)*, pages 21–35, 2008.

[30] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

[31] P. Zave and Jackson M. Four dark corners of requirements engineering. *Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

[32] T. Zheng, M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, 2008.