

Continually verify self-adaptation decisions taken by critical software in response to changes in the operating environment.

BY RADU CALINESCU, CARLO GHEZZI,
MARTA KWIATKOWSKA, AND RAFFAELA MIRANDOLA

Self-Adaptive Software Needs Quantitative Verification at Runtime

SOFTWARE IS BECOMING the backbone of modern society. Most human activity is either software-enabled or managed entirely through software, with examples ranging from health care and transportation to commerce and manufacturing. All increasingly

» key insights

- Human activity increasingly relies on software being able to make self-adaptation decisions on the fly.
- Offline approaches to verifying correctness before software deployment must be accompanied by continual online verification of the software's self-adaptation decisions.
- Quantitative verification at runtime supports continual re-verification of key requirements of self-adaptive software.

reflect one common requirement—the ability to adapt continuously in response to changes in application objectives and the environment in which the software operates. This reflects the vision of autonomic computing in which systems respond to change by evolving in a self-managed manner while running and providing service.^{4,9,20}

Dependability is another key requirement. As software use increases in business-critical and safety-critical applications, so, too, does the adverse

effect of unreliable or unpredictable software. Damaging effects, from loss of business to loss of human life, are no longer uncommon and must be addressed.

The requirements of adaptiveness and dependability are traditionally the concern of different research communities, with researchers involved in autonomic computing developing adaptive software systems for the past decade.^{19,30} In contrast, several mathematically based modeling and analysis techniques are used to improve software dependability, performance, and operating cost (such as energy consumption). Techniques include model checking¹⁰ and quantitative verification, a mathematically based technique for establishing the correctness, performance, and reliability of systems exhibiting stochastic behavior.²¹ They prevent errors from reach-

ing the software implementation or at least remove them when a new version of the software is deployed.

The only way to achieve such dependable software adaptation is to unite autonomic computing and mathematically based modeling and analysis techniques. Quantitative verification and model checking must also be used at runtime to predict and identify requirement violations, as well as to plan the adaptation steps necessary to prevent or recover from violations and obtain irrefutable proof the reconfigured software complies with its requirements. Software tools implementing flexible and low-overhead variants of both techniques must run automatically to support all stages of the adaptation process. The result is software capable of both self-adaptation to changes in its operating environment and continual verifica-

tion of its requirements compliance.

Here, we explore this new self-adaptation paradigm, explaining how quantitative verification can extend its operation to runtime. We then outline a range of complementary approaches that use formal verification techniques in runtime scenarios. Looking ahead, we present the main research challenges that must be addressed to make formal verification at runtime efficient and effective.

Reference Framework

Software evolution has been recognized as a distinctive feature since the early 1970s, most notably by Belady and Lehman.²⁴ Evolution is perhaps the most important feature distinguishing software from the other artifacts produced by humans. To shed light on software evolution, we refer to Zave's and Jackson's seminal work on requirements³¹ in which a clear distinction was made between the world and the machine. The machine is the system to be developed through software; the environment is the portion of the world that is to be affected by the machine (see Figure 1). The ultimate purpose of building a machine is always found in the world; requirements are statements on the desired phenomena in the world and should not refer to phenomena within the machine that concern only implementation. Some world phenomena are shared with the machine, controlled either by the world and observed by the machine or by the machine and observed by the world. A specification (for the machine) is a prescriptive statement of the relationships among

Figure 1. The world and the machine.

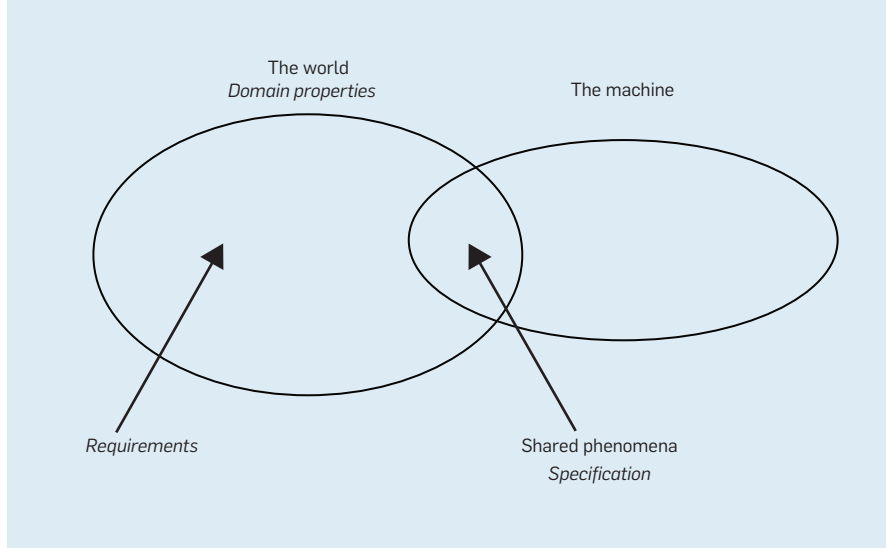
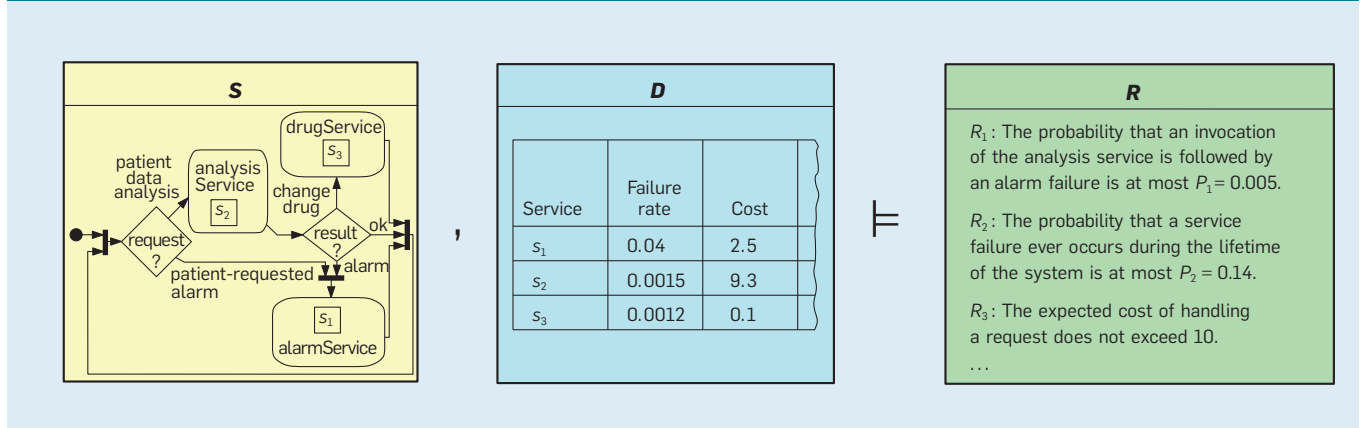


Figure 2. A medical-assistance application with specification S , domain assumptions D , and requirements R that satisfy Equation 1 in the main text.



shared phenomena that must be enforced by the system to be developed.


In developing a machine, software engineers must first derive a specification from the requirements and so must understand the relevant assumptions to be made about the environment in which the machine is expected to work, namely those affecting achievement of desired results; these assumptions are typically called domain knowledge; Zave and Jackson³¹ said it this way: “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 software engineers to prove (through the machine) they are able to achieve the desired requirements. Now let R and S be (prescriptive) statements describing the requirements and the specification in some formal notation, respectively, and let D be the (descriptive) formal statements specifying the domain assumptions. If S and D are satisfied and consistent, then a software engineer should be able to prove R also holds


$$S, D \models R. \quad (1)$$

Figure 2 outlines how this formalism applies to a simplified version of a medical-assistance system from Calinescu et al.⁵ The specification S , domain assumptions D , and requirements R of the system satisfy Equation 1. The specification S describes a service-based implementation of the medical-assistance system, including the ability to analyze patient data (provided by service s_2) or send a patient-requested alarm (service s_1). If service s_2 is invoked, 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 service costs s_1 , s_2 , and s_3 . The requirements R for the application include reliability-related requirements, defining, say, the maximum tolerated probability of failure for a specific sequence of service invocations.

Domain assumptions play a fundamental role in building systems that



The ultimate purpose of building a machine is always found in the world.



satisfy requirements. Engineers must know in advance the workings of the environment in which their software will be embedded, since the software is able to achieve the expected goals under only certain assumptions of the behavior of the domain described by D . Should these assumptions be invalidated, the software developed will most likely fail to satisfy its requirements.

Software evolution deals with changes affecting the machine, or specification S , that then cause changes in the implementation. Software evolution is triggered by a violation of the correctness criterion in Equation 1 discovered after the software is released. This violation may occur for any of three reasons:

- ▶ The implemented machine does not satisfy the specification;
- ▶ The behavior of the environment diverges from the domain assumptions D made when the specification was devised; and
- ▶ The requirements R do not capture the goals software users wish to achieve in the world.

A response to these changes is traditionally handled by modifying the software offline during a maintenance phase. The first reason corresponds to corrective maintenance. The second corresponds to adaptive maintenance; that is, S must be changed to satisfy the requirements under the newly discovered domain properties. And the third 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 software users. Because maintenance is an offline activity, software is returned to the development stage where the necessary changes are analyzed, prioritized, and scheduled. Changes are then handled by modifying the application's specification, design, and implementation. The evolved system is then verified, typically through some kind of regression testing, and redeployed.

Offline maintenance does not meet the needs of emerging application scenarios in which systems must run continuously and be capable of adapting autonomously the moment the need for change is detected. Here, we are interested in changes in the envi-

ronment D , using the term “self-adaptive software” to indicate that software has autonomous capabilities through which it tries to satisfy Equation 1 as changes to D are detected. These changes are typically due to one of two factors:

- High uncertainty about the behavior of the environment when the application is developed; and
- High variability in the behavior of the environment as the application runs.

Here, we focus mainly on system properties that can be expressed quantitatively and require quantitative verification (such as reliability, performance, and energy consumption); software must guarantee requirements increasingly expressed in terms of these properties and that are heavily influenced by the way the environment behaves, so environmental assumptions are increasingly crucial to software engineering; for example, assumptions concerning user-behavior profiles may affect overall system performance.

Self-adaptation can also be explained with reference to autonomic computing’s use of a monitor-analyze-plan-execute, or MAPE, closed control loop²⁰ to achieve self-management in computer systems. The four stages of the MAPE loop are enabled by knowledge combining assumptions D and specification S . This knowledge, updated continually through environment and system monitoring, helps analyze whether the user-specified requirements R continue to be satisfied. When they are no longer satisfied, appropriate system changes are planned and executed automatically.

Formal verification techniques like quantitative verification and model checking can provide the support required to integrate flexibility achieved through adaptation with dependability for critical software systems across the stages of the MAPE loop. We explore this support in the next section for quantitative verification and later for a range of related software modeling, specification, and analysis techniques.

Quantitative Verification at Runtime

Quantitative verification is a math-

ematically based technique for analyzing the correctness, performance, and reliability of systems exhibiting stochastic behavior.²¹ Technique users define a finite mathematical model of a system and analyze the model’s compliance with system requirements that are expressed formally in temporal logics extended with probabilities and costs/rewards; example requirements established through this analysis 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. Figure 3 outlines the quantitative verification of reliability requirements using discrete-time Markov chains, or DTMCs, to express specification S and domain assumptions D , and probabilistic computation tree logic, or PCTL, to formalize requirements R . Quantitative verification of performance requirements can be performed through complementary formalisms (such as continuous-time Markov chains, or CTMCs, and continuous stochastic logic, or CSL), and cost-related requirements can be verified through variants of these formalisms augmented with costs/rewards.²¹

Quantitative verification at runtime can support three stages of the software-adaptation process:

Monitoring. Precise, rigorous modeling of domain assumptions D (see Figure 4) is achievable by 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 its behavior; for example, for the DTMC in Figure 4, this component can update the service failure rates x , y , and z in line with the observed service behavior through the Bayesian learning methods introduced by Calinescu et al.⁶ and Epifani et al.¹² Likewise, the parameters of the CTMCs typically used to model performance-related aspects of software systems can be updated through Kalman filter estimators.³²

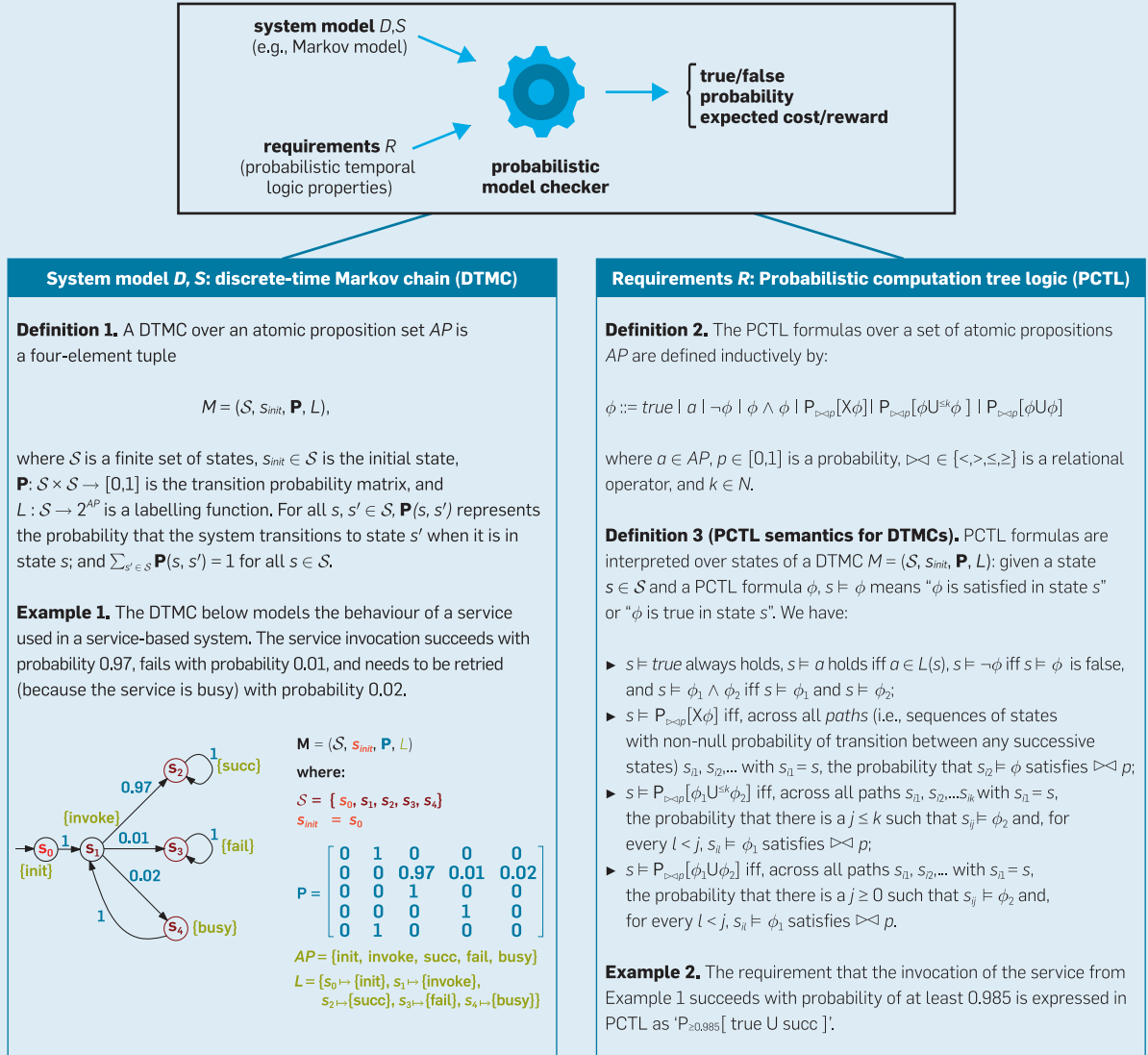
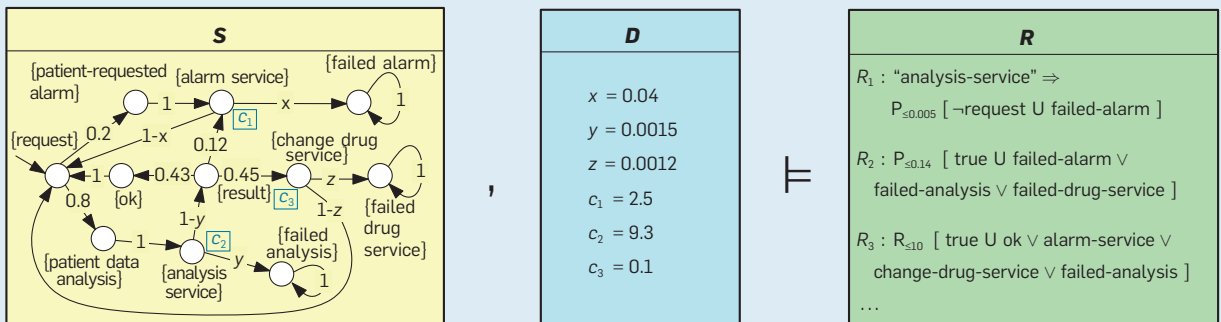
Analysis. A quantitative verification tool can be invoked automatically to detect (sometimes predict) requirement violations. Violation detection depends on the tool verifying the formally specified requirements R against the quantitative model ob-

tained by combining specification S with updated domain assumptions D from the monitoring stage. Figure 4 outlines the formalization of the relation $S, D \models R$ that enables the medical-assistance system mentioned earlier; in it, the specification S is modeled as a DTMC, comprising states for all possible system configurations (represented as circles) and state transitions (represented as edges annotated with the probabilities of the associated transitions). The domain assumptions D are variables that parameterize the model, reflecting the fact that service failure rates and costs may vary in real-world systems. Finally, the requirements R are expressed in probabilistic computation tree logic extended with the rewards operator R for models annotated with costs.

When a requirement $r \in R$ is no longer satisfied by the updated model, two scenarios are possible: the observation that triggered the model update was caused by observing system operations related to r , so the violation of requirement r is detected; and the updated model that does not satisfy r was obtained by observing system operations unrelated to this requirement, so the violation of requirement r is predicted; for example, an observed failure of the alarm service from the system in Figure 2 might 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 either of two events: an abnormal result from the analysis service, in which the analysis detects the violation of R_1 ; or a patient request, in which the violation of R_1 is predicted.

Planning. This stage is carried out when the analysis stage finds requirements (such as response time, availability, and cost) are or will be violated; as discussed earlier, adaptive maintenance leading to appropriate updates of the specification S is necessary in such circumstances. Quantitative verification can support planning by suggesting adaptive maintenance steps, execution of which ensures the system continues to satisfy its requirements despite the changes identified in the monitoring phase; for example, suppose the medical-assistance system in Figure 2 could select its alarm

Figure 3. Quantitative verification of reliability requirements.

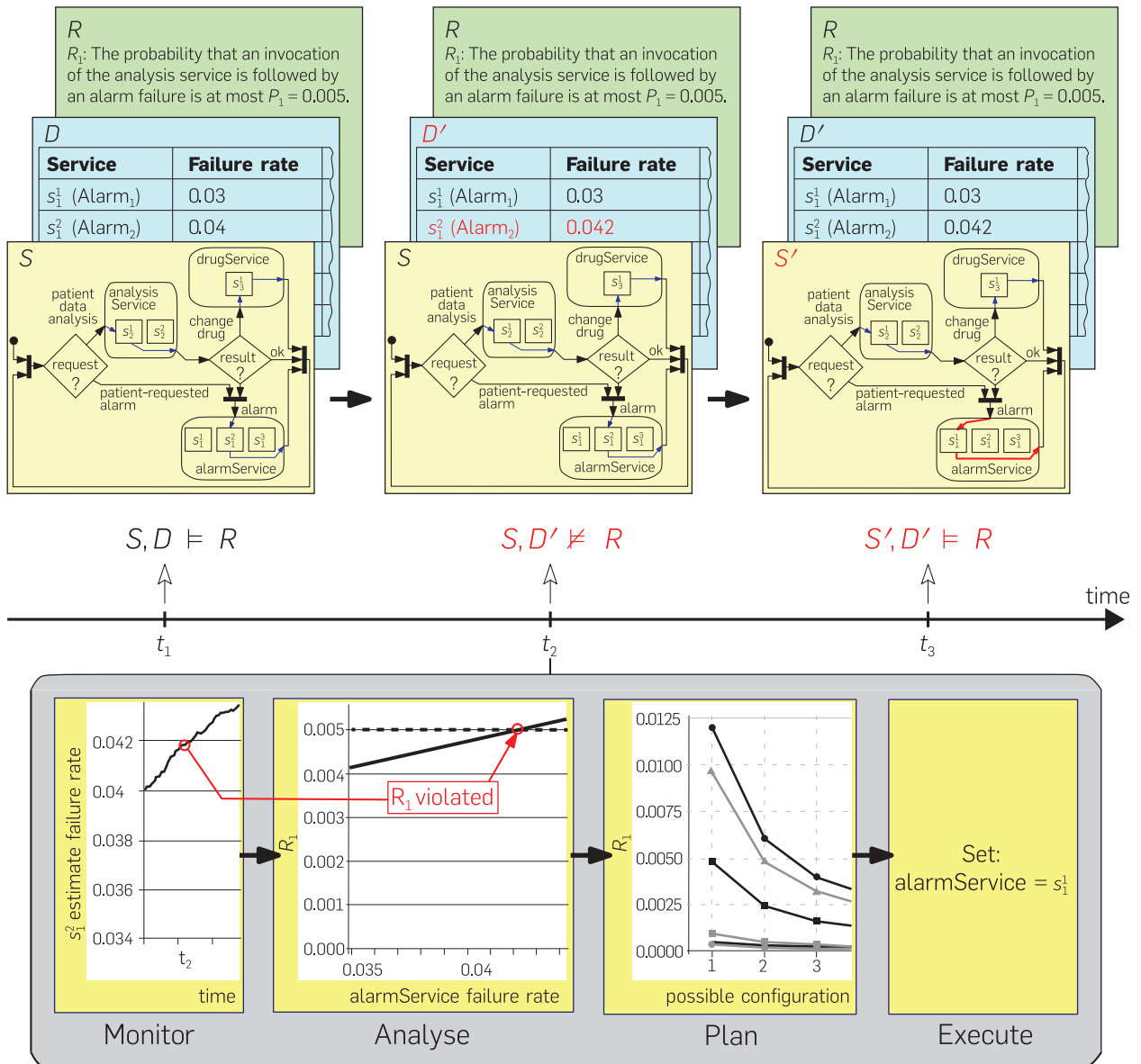

 Figure 4. Formalization of $S, D \models R$ for the medical-assistance application in Figure 2.


Self-Adaptive Service-Based Systems

Service-based systems are software applications built from loosely coupled services from multiple providers; used in various application domains, including e-commerce, online banking, and health care, they operate in environments characterized by frequent changes. As a result, their effectiveness depends increasingly on their ability to self-adapt. One way to devise self-adaptive service-based systems is to dynamically select the services that implement their operations from sets of functionally equivalent services associated with different levels of performance, reliability, and cost.

The figure here outlines a self-adaptive medical-assistance service-based system from Calinescu et al.³ and Epifani et al.¹² In the upper-left corner are the system's specification S , domain assumptions D , and requirements R at the initial time instant t_1 when the requirements are satisfied, or $S, D \models R$. However, as the failure rate of the alarm service used by the system, or s_1^1 , 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 : $S, D' \not\models R$ does not hold. To remedy this violation, the planning stage of the MAPE loop uses quantitative verification to select another service for the alarm operation. Accordingly, a new specification S' is employed to ensure the requirements are again satisfied at time instant t_3 : $S', D' \models R$.

Quantitative verification at runtime supports self-adaptation in service-based systems.



and analysis services dynamically from among the services provided by multiple third parties. Although functionally equivalent, these services are typically characterized by different levels of reliability, performance, and cost. A quantitative verification tool invoked automatically at runtime supports such dynamic service selection by establishing which combinations of alarm and analysis services, or specifications S , satisfy the requirements R at each time instant (see the sidebar “Self-Adaptive Service-Based Systems”).

We used the probabilistic model checker PRISM¹⁸ to validate the quantitative-verification-at-runtime approach described here in domains ranging from dynamic power management⁷ and data-center resource allocation⁸ to quality-of-service optimization in service-based systems.^{5,12} Success in these projects suggests that employing quantitative verification in runtime scenarios can augment software systems with self-adaptation capabilities in predictable ways.

Using Markovian models at a carefully chosen level of abstraction enabled these adaptive systems to operate with acceptable overheads for small- and medium-size systems. Scaling to larger systems requires faster runtime-verification algorithms; our recent research into devising such algorithms, exploiting the fact that the system model and verified requirements typically undergo only small changes from one adaptation step to the next, shows great promise.^{13,22}

Filieri et al.¹³ showed it is possible to pre-compute the probabilities associated with reliability-related requirements of a software system as symbolic expressions parameterized by domain assumptions; for 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

the medical-assistance 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 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 system size.

The approach taken by Kwiatkowska et al.²² achieves similar improvement through an incremental technique for verifying Markov decision processes (subsuming DTMCs discussed earlier) for the case where the probability value could vary at runtime. This approach exploits the fact that small changes in the model being verified often affect only a small subset of its strongly connected components, or SCCs. By reusing the verification results associated with the SCCs unaffected by change from one adaptation step to the next, the approach substantially reduces the computation cost of re-verifying the requirement. A symbolic implementation of the approach by Kwiatkowska et al.²² was shown to reduce the verification time by up to two orders of magnitude.

These scalable-verification approaches enable quantitative verification at runtime to develop larger adaptive software systems than was previously possible.

Related Work

For the past decade, several research communities have contributed toward integration of formal verification techniques into the runtime-software-adaptation process, with their results complementing our own work on quantitative verification at runtime; for example, Rushby’s work on runtime certification²⁹ emphasized the need for runtime configuration, arguing that any software reconfiguration at runtime must be accompanied by certification of the dependability of the new configuration. Building on Crow’s and Rushby’s previous research concerning a theory of fault detection, identification, and reconfiguration,¹¹ Rushby proposed

formal verification for achieving runtime certification, describing an enabling framework, including runtime use of “methods related to model checking.”²⁹ The range of correctness properties (such as safety and reachability) supported by this framework complements the reliability- and performance-related properties that can be managed through our quantitative verification at runtime.

Recent advances in using models at runtime provide additional evidence that runtime use of models is able to support software adaptation; for example, Morin et al.²⁶ described a method for developing adaptive software by predefining a set of system configurations, using aspect-oriented model reasoning to select the most suitable configuration at runtime. Different configurations may be associated with different quality-of-service properties or sets of supported services, an approach described as a “dynamic software product line.”²⁶ Similar results have been obtained through architectural models as a guide for the software-adaptation process.^{14,15} They employ general and user-defined constraint-verification techniques to change the architecture of a software system at a coarse level (such as by switching between two versions of a user interface). In contrast, runtime use of quantitative verification also supports fine-grain adaptation of system parameters (such as 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 be analyzed at runtime to establish in real time whether the software satisfies or violates correctness requirements expressed through various formalisms, including temporal logics,^{25,27} state machines,² regular expressions,¹ rule systems,³ and action-based contract languages.²³ However, unlike these approaches, quantitative verification at runtime supports software self-adaptation through quantitative verification and continuous monitoring of environment phenomena. Dynamic software composition (such as based on AI planning techniques²⁸) is another related research

area supporting adaptive reactions triggered by requirements violations.

Research Opportunities

Adaptive software development is an active research area that has produced a number of contributions beyond ad hoc practices. However, despite them, much remains to be done to support development of predictable adaptive software through a formal, systematic, disciplined approach. The remainder of the article elaborates on the main research areas where significant work is required to improve integration of formal verification techniques into software adaptation; the list is not exhaustive but reflects the key challenges encountered or foreseen in our own work and that of the research communities mentioned earlier.

We expect future software systems to be able to use discovery and model learning to operate in environments populated by active devices and appliances offering services and to be highly dynamic; for example, the context might change due to movement in space or to new services being deployed and discovered dynamically. These services (and the components providing them) might not know each other but still try to understand what they can do and possibly cooperate to achieve common goals. But how can a component learn what another component might offer, given different levels of visibility into the internals of the components? And how far can discovery and model learning go in the case of black-box visibility when only observations of a component's external behavior 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 This inference applies suitable learning strategies based largely on an assumption of regularity in the behavior of components. It has been tested successfully in the case of Java data abstractions,¹⁷ but further research is needed to make the approach general and practical.

Another area of research concerns integration of formal verification and self-adaptation, aiming to develop a repertoire of techniques that provides timely reaction to detected violations



**These scalable-
verification
approaches
enable quantitative
verification at
runtime to develop
larger adaptive
software systems
than was previously
possible.**



of the requirements. The strategies to follow in bringing this integration closer are very much domain- and application-dependent; for example, the techniques for speeding up runtime quantitative verification are justified when the time needed by the traditional variant of the technique is incompatible with the time needed for reaction. A catalogue of possible reaction strategies should be available at runtime, with one approach involving 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 for the software to perform self-adaptation at the model level, then replay model-driven development to derive an implementation through a chain of automatic transformations. If changes are anticipated at design time, they may be reified as variation points in the models; variations would then be generated dynamically to achieve adaptation. In the more challenging case of unanticipated changes, it might still be possible to devise a number of adaptation strategies and tactics the software can attempt at runtime.

Yet another research area addresses problems associated with new execution platforms (such as cloud computing). So far we have assumed that changes originate either in the requirements or in the domain assumptions, but with cloud computing, the infrastructure on which a machine works can 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 view of the platform and infrastructure running the software as services, too. Using a single abstraction to simultaneously reason about both the machine and the infrastructure may pave the way to “holistic” solutions. Self-adaptation cannot be seen at only the application level; the research community must conceive analysis techniques and identify solutions to drive self-adaptation of the overall system. Adaptations at the application level must consider the implications on the lower levels (such as component and infrastructure); conversely, these levels should provide a way for the applica-


tion to execute effectively. Adaptation becomes much more of an inter-level problem than a set of isolated intra-level solutions.

Cloud infrastructures also impose a shift from client-side, proprietary computing resources to shared resources. Web services forced software engineers to address the distributed ownership of their applications; the cloud is now forcing them to address the distributed ownership of the infrastructure used to run their applications. To some extent, problems associated with distributed ownership are already considered by application developers using services run and shared by others. Clouds complicate such problems significantly, with execution of one application competing against execution of another, turning self-verification and self-adaption into infrastructurewide requirements.

Conclusion

We discussed the runtime use of quantitative verification and model checking as ways to obtain dependable self-adaptive software. Our experience on a range of projects shows that quantitative verification at runtime can support software adaptation by identifying and, sometimes, predicting requirement violations; supporting rigorous planning of the reconfiguration steps self-adaptive software employs to recover from such requirement violations; and providing irrefutable proof the selected reconfiguration steps are correct. The result is software supporting not only automated changes but their continual formal analysis to verify that software continues to meet requirements as it evolves.

Acknowledgments

This research was partially funded by the European Commission Programme IDEAS-ERC, Project 227977-SMScom, by the U.K. 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. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. 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* (San Diego, Oct. 16–20). ACM Press, New York, 2005, 345–364.
2. Barringer, H. and Havelund, K. A Scala DSL for trace analysis. In *FM 2011: Formal Methods, Volume 6664 of Lecture Notes in Computer Science*, M. Butler and W. Schulte, Eds. Springer, Berlin/Heidelberg, 2011, 57–72.
3. Barringer, H., Havelund, K., Rydeheard, D., and Groce, A. Rule systems for runtime verification: A short tutorial. In *Run-time Verification, Volume 5779 of Lecture Notes in Computer Science*, S. Bensalem and D. Peled, Eds. Springer, Berlin/Heidelberg, 2009, 1–24.
4. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H.M., Litoiu, M., Müller, H.A., Pezze, M., and Shaw, M. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems, Volume 5525 of Lecture Notes in Computer Science*. Springer, Berlin/Heidelberg, 2009, 48–70.
5. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., and Tamburrelli, G. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering* 37, 3 (May–June 2011), 387–409.
6. Calinescu, R., Johnson, K., and Rafiq, Y. Using observation ageing to improve Markovian model learning in QoS engineering. In *Proceedings of the second ACM/SPEC International Conference on Performance Engineering* (Karlsruhe, Germany, Mar. 14–16). ACM Press, New York, 2011, 505–510.
7. Calinescu, R. and Kwiatkowska, M. CADs*: Computer-aided development of self-* systems. In *Fundamental Approaches to Software Engineering, Volume 5503 of Lecture Notes in Computer Science*, M. Chechik and M. Wirsing, Eds. Springer, Berlin/Heidelberg, 2009, 421–424.
8. Calinescu, R. and Kwiatkowska, M. Using quantitative analysis to implement autonomic IT systems. In *Proceedings of the 31st International Conference on Software Engineering* (Vancouver, Canada, May 16–24). IEEE Computer Society Press, Washington, D.C., 2009, 100–110.
9. Cheng, B.H. et al. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, B.H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer-Verlag, Berlin/Heidelberg, 2009, 1–26.
10. Clarke, E.M. and Lerda, F. Model checking: Software and beyond. *Journal of Universal Computer Science* 13, 5 (May 2007), 639–649.
11. Crow, J. and Rushby, J. *Model-Based Reconfiguration: Diagnosis and Recovery*. NASA Contractor Report 4596. NASA Langley Research Center, Hampton, VA, May 1994; work performed by SRI International.
12. Epifani, I., Ghezzi, C., Mirandola, R., and Tamburrelli, G. Model evolution by run-time adaptation. In *Proceedings of the 31st International Conference on Software Engineering* (Vancouver, Canada, May 16–24). IEEE Computer Society Press, Washington, D.C., 2009, 111–121.
13. Filieri, A., Ghezzi, C., and Tamburrelli, G. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering* (Honolulu, May 21–28). IEEE Computer Society Press, New York, 2011, 341–350.
14. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjorven, E. Using architecture models for runtime adaptability. *IEEE Software* 23, 2 (Mar. 2006), 62–70.
15. Garland, D. and Schmerl, B.R. Using architectural models at runtime: Research challenges. In *Proceedings of the first European Workshop on Software Architecture, Volume 3047 of Lecture Notes in Computer Science*, F. Ogundo, B. Warboys, and R. Morrison, Eds. (St. Andrews, Scotland, May 21–22). Springer-Verlag, Berlin/Heidelberg, 2004, 200–205.
16. Ghezzi, C., Mocci, A., and Monga, M. Synthesizing intentional behavior models by graph transformation. In *Proceedings of the 31st International Conference on Software Engineering* (Vancouver, Canada, May 16–24). IEEE Computer Society, Washington, D.C., 2009, 430–440.
17. Ghezzi, C., Mocci, A., and Salvaneschi, G. 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*. Springer, Berlin/Heidelberg, 2010, 233–247.
18. Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. PRISM: A tool for automatic verification of probabilistic systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Volume 3920 of Lecture Notes in Computer Science*, H. Hermanns and J. Palsberg, Eds. Springer, Berlin/Heidelberg, 2006, 441–444.
19. Huebscher, M.C. and McCann, J.A. A survey of autonomic computing: Degrees, models, and applications. *ACM Computing Surveys* 40, 3 (Aug. 2008), 1–28.
20. Kephart, J.O. and Chess, D.M. The vision of autonomic computing. *Computer* 36, 1 (Jan. 2003), 41–50.
21. Kwiatkowska, M. Quantitative verification: Models, techniques, and tools. In *Proceedings of the Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium Foundations of Software Engineering* (Dubrovnik, Croatia, Sept. 3–7). ACM Press, New York, 2007, 449–458.
22. Kwiatkowska, M., Parker, D., and Qu, H. Incremental quantitative verification for Markov decision processes. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks* (Hong Kong, June 27–30). IEEE Computer Society, Los Alamitos, CA, 2011, 359–370.
23. Kyas, M., Prisacariu, C., and Schneider, G. Run-time monitoring of electronic contracts. In *Proceedings of the Sixth International Symposium on Automated Technology for Verification and Analysis* (Seoul, Oct. 20–23). Springer-Verlag, Berlin/Heidelberg, 2008, 397–407.
24. Lehman, M.M. and Belady, L.A., Eds. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, 1985.
25. Leucker, M. and Schallhart, C. A brief account of runtime verification. *Journal of Logic and Algebraic Programming* 78, 5 (May 2009), 293–303.
26. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., and Solberg, A. Models@run.time to support dynamic adaptation. *Computer* 42, 10 (Oct. 2009), 44–51.
27. Pnueli, A. and Zaks, A. PSL model checking and runtime verification via testers. In *Proceedings of the 14th International Symposium on Formal Methods* (Hamilton, Canada, Aug. 21–27). Springer-Verlag, Berlin/Heidelberg, 2006, 573–586.
28. Rao, J. and Su, X. A survey of automated Web service composition methods. In *Semantic Web Services and Web Process Composition, Volume 3387 of Lecture Notes in Computer Science*, J. Cardoso and A. Sheth, Eds. Springer, Berlin/Heidelberg, 2005, 43–54.
29. Rushby, J.M. Runtime certification. In *Proceedings of the Eighth International Workshop on Runtime Verification, Volume 5289 of Lecture Notes in Computer Science*, M. Leucker, Ed. (Budapest, Mar. 30). Springer-Verlag, Berlin/Heidelberg, 2008, 21–35.
30. Salehie, M. and Tahvildari, L. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4, 2 (May 2009), 14:1–14:42.
31. Zave, P. and Jackson, M. Four dark corners of requirements engineering. *Transactions on Software Engineering and Methodology* 6, 1 (Jan. 1997), 1–30.
32. Zheng, T., Woodside, M., and Litoiu, M. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering* 34, 3 (May–June 2008), 391–406.

Radu Calinescu (Radu.Calinescu@york.ac.uk) is a senior lecturer in large-scale complex IT systems in the Department of Computer Science, University of York, U.K.

Carlo Ghezzi (Carlo.Ghezzi@polimi.it) is a professor and chair of software engineering in the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, and president of Informatics Europe.

Marta Kwiatkowska (Marta.Kwiatkowska@cs.ox.ac.uk) is a professor of computing systems and fellow of Trinity College, University of Oxford, U.K.

Raffaella Mirandola (mirandola@elet.polimi.it) is an associate professor in the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy.