

The Disappearing Boundary Between Development-time and Run-time

Luciano Baresi and Carlo Ghezzi

Politecnico di Milano, Dipartimento di Elettronica e Informazione – DeepSE Group

Piazza L. da Vinci 32, 20133 Milano, Italy

luciano.baresi@polimi.it, carlo.ghezzi@polimi.it

ABSTRACT

Modern software systems are increasingly embedded in an open world that is constantly evolving, because of changes in the requirements, in the surrounding environment, and in the way people interact with them. The platform itself on which software runs may change over time, as we move towards cloud computing. These changes are difficult to predict and anticipate, and their occurrence is out of control of the application developers. Because of these changes, the applications themselves need to change. Often, changes in the applications cannot be handled off-line, but require the software to self-react by adapting its behavior dynamically, to continue to ensure the desired quality of service. The big challenge in front of us is how to achieve the necessary degrees of flexibility and dynamism required by software without compromising the necessary dependability.

This paper advocates that future software engineering research should focus on providing intelligent support to software at run-time, breaking today's rigid boundary between development-time and run-time. Models need to continue to live at run-time and evolve as changes occur while the software is running. To ensure dependability, analysis that the updated system models continue to satisfy the goals must be performed by continuous verification. If verification fails, suitable adjustment policies, supported by model-driven re-derivation of parts of the system, must be activated to keep the system aligned with its expected requirements.

The paper presents the background that motivates this research focus, the main existing research directions, and an agenda for future work.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.4 [Software Engineering]: Software/Program Verification—*Formal methods*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...\$10.00.

General Terms

Design, Reliability, Verification

Keywords

Runtime Support, Self-adaptation, Service-oriented Systems

1. INTRODUCTION

The clear separation between development-time and run-time is one of the dogmas of traditional computer science. Traditional software engineering research has been mostly focused on development-time, investigating how the software development process can be understood, improved, and automated. The ultimate goal, of course, is producing quality software; that is, software that correctly performs its functionality and achieves the goals the users expect from it. Methods, techniques, and tools have been developed to ensure correctness and satisfaction of all functional and non-functional requirements before software is delivered to users and put in operation, to minimize the chance that the software misbehaves while in operation. In the state of the practice, almost no or little attention has instead been placed on supporting software at run-time, and only in recent years software engineering research started focusing on these issues. The implicit traditional assumption is that the software can only be observed at run-time, but almost nothing can be done to affect it or modify it while it is being enacted. The wealth of methods and tools that are used at development-time to forge the software have no more use when the software enters the run-time stage. If problems are found that need corrective actions, error reports and change requests are filed by users and then prioritized by software developers to be handled off-line.

This framework is changing today and will change more in the future. The clear separation between development-time and run-time is blurring and may disappear in the future for relevant classes of applications. In this paper, we briefly motivate why and how this is happening (Section 2). We then discuss in Section 3 some research that is being performed both by us and by other researchers to tackle some of the problems arising in these contexts. Finally, in Section 4 we outline an agenda for future work and in Section 5 we draw some conclusions.

2. CONTINUOUS CHANGE

Software *evolution* has been recognized as key distinctive feature since the early 1970's by many researchers, and most

notably by Belady and Lehman [11]. Indeed, perhaps evolution is the most important aspect that distinguishes software from other artifacts produced by humans. To understand the deep causes of evolution, we can refer to the seminal work on requirements by Jackson and Zave [14], which illustrates the relation between the *machine* and the *world*.

The machine is the system to be developed; the world (the environment) is the portion of the real-world affected by the machine. The ultimate purpose of the machine is always to be found in the world. *Requirements* thus refer to the desired phenomena occurring in the world, as opposed to phenomena occurring inside the machine. Some of such 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. Finally, it is important to understand the set of relevant assumptions that can be made about the environment in which the machine is expected to work, which affect the achievement of the desired results. This is also called *domain knowledge*. Quoting from [14]: “*The primary role of domain knowledge is to bridge the gap between requirements and specifications*”.

In fact, if R and S are the prescriptive statements that formalize the requirements and the specification, respectively, and D are the descriptive statements that formalize the domain assumptions, it is necessary to prove that

$$S, D \models R$$

that is, S entails satisfaction of the requirements R in the context of the domain properties D . Thus D plays a fundamental role in establishing the requirements. We need to know upfront how the environment in which the software is embedded works, since the software can achieve the expected requirements only based on the assumptions on the behavior of the domain described by D . Should these assumptions be invalidated, as a consequence the requirements might be violated. Environment assumptions may be invalidated for two reasons: either because the domain analysis was flawed or because the environment has changed, and the assumptions that were made earlier are no longer valid.

Software evolution refers to changes that may be made to the machine to respond to changes in the requirements and/or in the environment (we assume the implementation to be correct, i.e., no failures are due to errors in the machine implementation). Changes in the requirements mostly fall under the traditional *perfective maintenance* category, and may be dictated by changes in the business goals of organizations or new demands by users of older versions of an application. Environmental changes instead affect the assumptions that ensure the satisfaction of the requirements. They may represent organizational assumptions or conditions on the physical context in which the machine is embedded. If these assumptions are invalidated, the software must undergo what is traditionally called *adaptive maintenance*.

The management of changes in the maintenance phase clearly indicates that traditional software evolution is an off-line phase. Software returns in the development stage, where changes are analyzed, prioritized, and scheduled. Changes are then handled by modifying the design and implementa-

tion of the application. The evolved system is then verified, typically via some kind of regression testing.

This lifecycle does not meet the requirements of many emerging application scenarios, which are subject to continuous changes in the requirements and in the environment, and which require rapid adaptation to such changes. Increasingly, both recognition of and reaction to changes must be managed automatically by the running system, and off-line human intervention must be limited to only special cases. Run-time adaptation must be achieved seamlessly, as the application is running and providing service.

There are already numerous examples of existing systems where these requirements hold. At the business level, information systems are increasingly built through service units and their dynamic integration. Service units may be supplied by different providers, crossing enterprise boundaries. The service network is dynamic, since changes in business conditions may require automatic adaptation of the network. At another extreme, pervasive computing systems are also characterized by continuous, dynamic changes, mostly due to contextual changes. A typical context change is due to mobility, which may expose the application to unexpected changes in, for example, the assumptions made on certain features provided by other devices in the environment with which the application interacts.

All the examples of systems we refer to provide their functionality by relying on other applications that exist in the environment. Such applications, which we call *services*, differ from components in traditional component-based software systems. Services, in fact, are run autonomously remotely. They may be discovered dynamically and then be invoked by other applications to have some task executed. Normally, these services belong to different stakeholders, who have full responsibility over their evolution, deployment, and execution. They may commit to satisfying a certain *quality of service*, but client application cannot have full trust into it. Systems built out of services, called *service-oriented systems*, are therefore characterized by *distributed ownership*. No single stakeholder is in control of the whole system. Rather, new systems are built out of parts that may evolve dynamically and autonomously. Service-oriented systems are an increasingly important —though not exclusive— class of systems for which dynamic evolution and adaptation is crucial. In this paper we deliberately focus on them.

The Jackson-Zave framework can help us better understand the nature of changes in R and D and the way they can be handled. First, to be more precise in the terminology, we may use the term *evolution* to indicate changes that require human intervention and off-line modification of the software. The term *adaptation* may instead be used to indicate changes that the software itself can handle autonomously. Changes in R mostly lead to evolution: they can only be understood and handled by humans. Instead changes in D may largely be handled by adaptation.

Following [10], and focusing on the specific example of an e-commerce application built by integrating (*orchestrating*) a number of existing Web services, we can decompose the set of domain properties D upon which requirements rely into two main disjoint subsets, D_u and D_s . The set D_s collects all the *assumptions on the external services* invoked by the application, which describe what the composite application to be developed expects from them to achieve its own goals. If S_1, S_2, \dots, S_n are the required specifications of these ex-

ternal services, $D_s = S_1 \cup S_2, \dots \cup S_n$. D_u instead denotes the *assumptions on usage profiles*. It consists of properties that characterize how the final integrated system is expected to be used by its clients. A possible example of a usage profile is: “*The probability that users buying technical books also order express shipping instead of normal shipping is 0.6*”. D_s and D_u describe domain assumptions made by the software engineer at design-time. Proper design must ensure that the application satisfies the requirements, under the assumption that D_s and D_u correctly characterize the behavior of the environment. However, these assumptions are subject to high uncertainty and may change dynamically. As we discuss hereafter, changes in D can often be detected automatically by suitable *sensors* that monitor the environment and can trigger strategies to achieve self-adaptation.

3. EXISTING WORK

Existing work addressing the issues outlined in Section 2 falls in the areas of self-adaptive systems¹ and autonomic systems², currently investigated by active international research communities. Existing approaches may be organized according to the *Monitor-Analyze-Plan-Execute* (MAPE) loop. To achieve self-adaptation, systems must be able to monitor the possible sources of change, through suitable (abstract) sensors which may detect changes in the environment that require suitable reaction. To understand if a relevant change occurred, the data collected by the monitor must be analyzed. As a result of analysis, a change plan must be identified, and eventually executed.

Monitoring has been an active research field for years [6] and the advent of services has motivated a new thread of works. The distributed ownership of service compositions has imposed constraints on what can be monitored and on how it can be done. The actual deployment of server-side probes to assess the vital parameters of services is not feasible, and most of the monitoring approaches (e.g., [13, 12, 2]) only assess the values of interest as perceived by the application. Besides this, monitoring approaches can be characterized by the information they collect and by their impact on the execution of the actual business logic. Some approaches probe the messages exchanged between the parties [13] and can only provide low-level and domain-independent data (e.g., the actual throughput or the mean time between a request and its answer). In contrast, other proposals work at a higher level and provide users with business-related data [12]. Orthogonally, some approaches work in parallel with the business logic, do not interfere with it, but can only be used for *off-line* analyses. Other approaches are more intrusive and intertwine the execution of the business logic with the collection (and analysis) of data. The execution becomes slower, but the detection of anomalies, along with the enactment of corrective actions, becomes timely and more focused.

As for analysis, research has been focusing on two main issues. One is *run-time verification*³, possibly achieved by

keeping *models at run-time*⁴ [4], which may be used to perform continuous analysis that the (model of the) application continues to behave satisfactorily. The other concerns data analysis, which may apply *machine learning* techniques to transform the data produced by the monitors into information that may be used to verify the model at run-time.

If analysis highlights problems, planning is often carried out through policies and event-condition-action rules [5]. The event is triggered by the analysis, possible conditions help further identify the reaction, and the action is a pre-computed activity (or set of activities), which aim at keeping the application on track. Possible adaptations span from the simple re-execution of the interaction with the misbehaving partner to the re-selection of the partner service. Some approaches (e.g. [3]) have also tried to apply actual planning techniques to adapt the application by re-computing a new service composition able to carry out the task because of a new goal or a new context (i.e, set of services), but their suitability as “general-purpose solution” is still to be fully proved.

Our own past work has been focusing on several of the aforementioned aspects. We proposed an *assertion-based* monitoring and recovery solution for service compositions [2]. We developed two special-purpose languages, called WSCoL (Web Services Constraint Language) and WSReL (Web Service Recovery Language), to let the designer collect relevant data, specify punctual assertions on the interactions between the composition and its partner services, and identify corrective/adaptive actions if needed. WSCoL lets the designer collect data from the composition itself, the environment, and previous executions; assertions can predicate on both functional and non-functional properties. WSReL supports the definition of articulated recovery strategies by composing atomic activities (like changing the partner service, retrying the interaction, or rolling back the execution). The evaluation of these assertions, along with the execution of associated recovery actions, is synchronous with respect to the execution of the business logic to provide the user with a focused and precise management tool. Its impact at run-time can be tuned by switching on and off the evaluation of the different constraints.

As for analysis, we have been focusing on functional, temporal, and quantitative probabilistic analysis to achieve continuous verification of evolving systems ([1, 7, 10]). In particular, to analyze non-functional requirements, such as performance or reliability, we explored the use of Markov models for the application. Models can be analyzed via probabilistic model checking and changes in the model parameters due to environment changes may be learnt by using a Bayesian approach, which may update the parameters based on data collected by monitors.

4. OPEN ISSUES

Adaptive software systems are increasingly built in practice. As discussed in the previous section, research in this area has also been active since several years, and several contributions have already been produced to move beyond the current ad-hoc practices. Much remains to be done to support development of adaptive systems via a comprehensive, systematic, and disciplined approach. Hereafter we elab-

¹International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).

²International Conference on Autonomic Computing (ICAC) or the International Conference on Self-Adaptive and Self-Organizing Systems (SASO).

³International Conference on Runtime Verification (RV)

⁴International Workshop of Models at Runtime (models@runtime).

orate on a number of key research areas. The list is not exhaustive, but simply reflects our current interests and research priorities.

A first area concerns *discovery* and *learning*. In the future, the environments in which we operate will be fully populated by active devices and appliances which will offer services. Services will come and go dynamically; they will meet each other and cooperate. They might not know each other, but may still try to understand what each can do and possibly cooperate to achieve common goals. How can this be achieved? How can a component learn about what another component offers? How can this be achieved depending on different levels of visibility into the internals of the components? For example, how far can we go in the case of *black-box* visibility, i.e., only observation of the external component's behavior? How can the observations of component be trusted? How can they be generalized? In this area we did some initial work which aims at inferring the functional behavior of a (stateful) component by observations of inputs and outputs at its API [8]. Our method, which applies suitable learning strategies, is largely based on an assumption of *regularity* of the components' behaviors. It has been tested quite successfully in the case of Java data abstractions [9], but more needs to be done to make the approach general and practical.

A second area concerns *run-time verification*. In our work so far we used at run-time the same models that were initially used to assess the application at design-time. In particular, for quantitative probabilistic properties, we used Markov chains and probabilistic model checking. Resorting to the same models guarantees that exactly the same analysis, with the same level of accuracy, is repeated at run-time. This, however, may clash with the requirements of execution time efficiency that may be required to allow the adaptation policies to react effectively (timely) to changes. Possible alternative solutions may be explored, which may simplify the properties to monitor at run-time. Alternatively, one might explore how to make model checking more efficient; for example, by making it incremental. Yet another approach (which we are currently exploring), consists of deriving closed formulae for the properties to be checked at run-time, where changeable values of environment data are represented as variables, whose values become known at run-time. The computation of such closed formulae from requirements properties requires expensive symbolic formula manipulations, which may however be performed at design-time. On the other hand, run-time checking is very efficient. It is still unclear how general the approach is; that is, the class of requirements assertions for which closed formulae may be derived.

A third area concerns *run-time self-adaptation*. This is a very active area of research, as we mentioned in the previous section. An interesting, and to the best of our knowledge, unexplored approach would be to address it in the model-driven framework. Since models are kept alive at run-time, once the need for adaptive reactions is identified, it would be useful to perform self-adaptation at the model level, and then re-play model-driven development to derive an implementation through a chain of automatic transformations. To do that, we would need a library of model adaptation strategies, and tactics to select them based on the required adaptation. This would be an interesting research area, where progress is needed. Progress is also needed in the automatic

chain of transformations that can derive the final implementation to be run as a target of the adaptive transformation process.

A fourth crucial area concerns the problems arising from new execution platforms, such as *cloud computing*. So far we assumed that changes are either in R or in D . But with the advent of cloud computing, also the infrastructure on which our machine works may change. If we exploit the full potential of the *service* paradigm, we must complement our usual application level (*software-as-a-service*) with the platform and infrastructure on which the software is run, and they both can be seen as services. The use of a single abstraction to reason on both the machine and the infrastructure may pave the ground to "holistic" solutions. Self-adaption cannot be seen at application level only, but we must deploy probes, conceive analysis techniques, and identify solutions able to self-adapt the system as a whole. Adaptations at application level must consider the implications on the lower levels, but conversely these levels provide the means to let the application execute properly. A given quality of service at application level may be a consequence of assumptions and decisions on the lower levels; adaptation becomes much more an inter-level problem than a set of isolated, and maybe cooperating, 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 infrastructures. Partially, this is already the case when we want to use components (services) run and shared by others, but clouds will enlarge the problem. The execution of one application will compete with the execution of others, turning self-verification and adaption into infrastructure-wide problems.

5. CONCLUSIONS

The paper identifies the need for intelligent support to software at run-time as one key direction for future research in software engineering. The boundaries between development-time and run-time are too rigid, and we already have many important applications that require more and more runtime adaption instead of traditional adaptive maintenance. The key enabler for this shift will be the transformation of models in runtime entities able to guide and coordinate changes and modifications. The main existing solutions in this direction and an agenda for the future complete the proposal.

Acknowledgments

This paper reflects the work being developed within the SM-Scom project, funded by the European Commission, Programme IDEAS-ERC, Project 227977 (www.erc-smscom.org).

6. REFERENCES

- [1] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of Web Service Compositions. *IET Software*, 1(6):219–232, 2007.
- [2] I. Baresi and S. Guinea. Self-supervising BPEL Processes. *IEEE Transactions on Software Engineering*, 2010. to appear.
- [3] P. Bertoli, M. Pistore, and Traverso P. Automated Composition of Web Services via Planning in

- Asynchronous Domains. *Artificial Intelligence*, 174(3-4):316–361, 2010.
- [4] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, pages 22–27, 2009.
- [5] M. Colombo, E. Di Nitto, and M. Mauri. SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In *Proceedings of the 4th International Conference on Service-Oriented Computing*, volume 4294 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 2006.
- [6] N. Delgado, A. Quiroz Gates, and S. Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] C. Ghezzi and G. Tamburrelli. Reasoning on Non-Functional Requirements for Integrated Services. In *Proceedings of the 17th International Requirements Engineering Conference*, pages 69–78. IEEE Computer Society, 2009.
- [11] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985.
- [12] M. Mahbub and G. Spanoudakis. A framework for requirements monitoring of service based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 84–93. ACM Press, 2004.
- [13] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceedings of the 17th International Conference on World Wide Web*, pages 815–824. ACM, 2008.
- [14] P. Zave and Jackson M. Four Dark Corners of Requirements Engineering. *Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.