Dynamically Adaptive Systems through Automated Model Evolution Using Service Compositions

Adina Mosincat, Walter Binder, and Mehdi Jazayeri

Faculty of Informatics, University of Lugano, Switzerland
{adina.diana.mosincat,walter.binder,mehdi.jazayeri}@usi.ch

Abstract. Runtime adaptability and delivered quality are two important concerns for every system. One way to achieve runtime adaptability is by specifying variants in the system model at design time to allow switching between runtime configurations. The fulfillment of system's quality requirements depends on parameters that can change at runtime. In order to meet its quality requirements, the system must be able to dynamically adapt to changes that affect the delivered quality. We outline our approach to enhance system adaptability through automatic evolution of the system model. Our approach periodically updates the model by re-evaluating the delivered quality based on runtime information. We use a service composition model to represent the system functional requirements and annotate it with delivered quality evaluations. We ensure system runtime adaptability by selecting the variant to execute at runtime based on the evolved model.

1 Introduction

Systems are increasingly required to feature autonomic capabilities. Systems that need to be continuously available and be able to change behavior depending on environmental conditions require a higher degree of autonomic capabilities. These systems must dynamically adapt to changes in the environment, switching between different runtime configurations without disrupting the running system. Software engineers can develop such dynamically adaptive systems by providing variants that determine the runtime configurations, respectively the execution paths of the system.

Quality requirements[1] are important concerns for every system. When reasoning about design decisions, software engineers must also ensure that the system will meet its quality requirements. The engineers face two issues:

1. The variants provided at design time are limited to the knowledge of the engineers and to their capability to foresee changes. Unanticipated changes in the environment or changes demanded by new user needs require modification and redeployment of the system.

[1] In this paper we use the term quality requirements for non-functional requirements, such as performance, reliability, and cost.

2. To estimate the fulfillment of the quality requirements, software engineers use parameters provided at design time that estimate the behavior of system components in a given environment. In a changing environment, parameter changes can result in quality degradation or violations of the system's quality requirements.

To address these issues, we automatically evolve the model of the system, periodically update it with observed quality information, and use the evolved model to drive the choice of the system runtime configuration. We present a framework for automated model evolution that uses a service composition model to represent the running system. The choice of the model allows us to leverage the advantages of service oriented architectures: flexibility, loose coupling, and ease of integration. Changes in our model are immediately incorporated in the implementation and runtime configuration. By using the model at runtime, our approach can easily integrate new variants and modify existing ones without disrupting the running system.

The Quality of Service (QoS) parameters of a service-oriented system depend on the QoS of its composing services. QoS estimates are expressed as guarantees, called service level objectives (SLO [1]), in service level agreements (SLA [1]). However, given the dynamic and distributed nature of service-oriented systems, QoS can vary in time and the system cannot enforce its composing services to meet their SLOs. Our framework monitors service execution and periodically re-evaluates the QoS parameters of the system based on monitoring information. It then updates the model with the new QoS values that are used for selecting the runtime configuration.

In this paper we outline our approach to enhance runtime adaptation capabilities for dynamically adaptive systems by updating the system model according to runtime information and using the updated model at runtime to drive the system runtime configuration. We give an overview of a possible implementation of our approach using service compositions and the BPEL standard.

2 Approach Overview

An important concept that we use in our approach is the *variant*. A variant represents one possible implementation of a system functional requirement. Variants can increase fault tolerance by specifying alternative ways of fulfilling a functional requirement. Take for example the case of the alarm function in a smart home system. The engineer specifies one variant of fulfilling the functionality by setting off the alarm, and a second variant by blinking the lights.

Fig. 1 presents the conceptual phases in our approach: (1) automatically evolving the system model based on runtime information; and (2) using the updated model at runtime to adapt the running system.

Initially, the delivered quality values of the system are estimated using parameters available at design time, such as QoS guarantees specified by SLAs. Due to system evolution and environmental changes, these parameters might change,

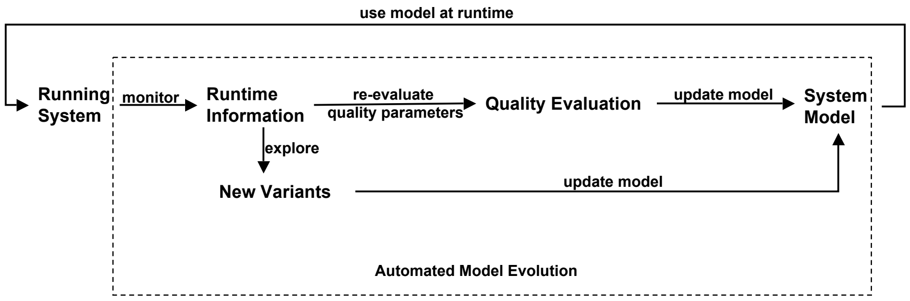

Fig. 1. Approach overview diagram

making the initially estimated quality values obsolete. By monitoring the running system, it is possible to gather information that reflects the current state of the environment. The quality values are re-estimated based on the monitoring information and the model is then updated according to the new estimations.

The algorithms used to estimate the quality values can help detect the system's inability to meet its quality requirements. The estimations can also help detect the cause of a quality degradation, such as a malfunctioning component. The system can improve the delivered quality and prevent violations of quality requirements by leveraging the estimations when selecting the variant to execute.

A second concern for model evolution are changes that provide a new variant, or that invalidate an existing variant. Consider for example the case of introducing a new component. New variants using the component become available. The model is updated by integrating the new discovered variants.

System adaptation is done by using the updated model to decide the system runtime configuration and the execution path for each functional requirement. To use this adaptation technique, a framework implementing our approach must ensure that implementation always conforms to the current model. The framework must provide a way to automatically implement the model changes into the running system without requiring user intervention or system interruption. The choice of the system model plays an important part in achieving the required autonomic capabilities.

3 A Framework Based on Service Compositions

This section presents a framework illustrating the approach introduced above and describes the system model and the components of the framework that play an important part in the automated model evolution and system adaptation.

3.1 Model

Fig. 2 presents the generation of the system model from the developer input model. In the input model the developer defines variability points, that is, functional (sub)requirements of the system which can have different variants. The

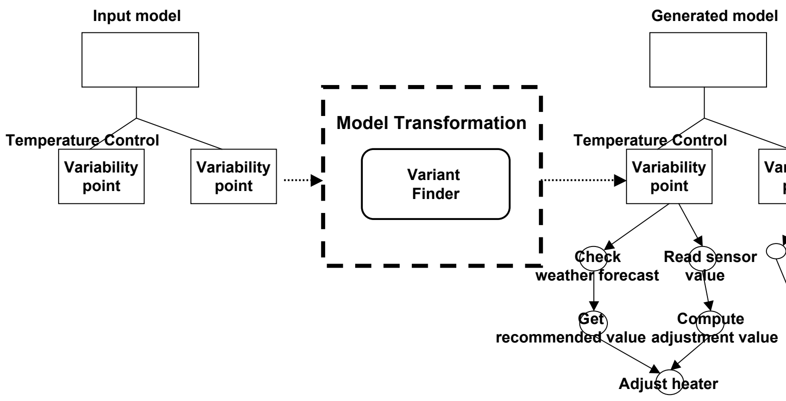

Fig. 2. Model transformation

Variant Finder generates the model of the running system from the input model by finding solutions to the variability points. A solution to a variability point is a variant.

The *Variant Finder* is an automated service composition engine that takes the variability point description and queries the service repository finding the set of service compositions that solve the requirement, such as [2,3]. The variability points must be expressed in a query language understood by the *Variant Finder*, and the services must be semantically annotated. In previous work we have introduced a query language for directories in support for automated service composition [4].

The input model must represent a system that can be implemented as a service composition. Currently we consider the input model to be a goal model with functional goals (functional requirements) as presented in [5]. A goal model is an AND/OR graph showing how higher-level goals are satisfied by lower-level ones (goal refinement) [6]. The AND-refinement link relates a goal to a set of subgoals that must be satisfied in order for the goal to be satisfied. A goal node can be OR-refined into multiple AND-refinements that each represent an alternative, i.e. the parent goal can be satisfied by satisfying the subgoals in any of the alternative AND-refinements. In our input model, the bottom subgoals must be queries in order to be able to automatically generate the system implementation. In this case, the whole system implementation is provided through automatic service composition.

The generated model is an annotated AND/OR graph in which the variability points are expanded with the variants found by the *Variant Finder*. A *variant node* is a node that is parent to at least one variant. All OR-link nodes and nodes corresponding to variability points in the input model are *variant nodes*. *Variant nodes* are annotated with information that is used at runtime to select the variant to execute. A *variant node* contains as data for each child variant a set of estimated QoS values and the deviation percentage from the required QoS. At model generation, the QoS values are computed based on the SLAs of the variant's composing services.

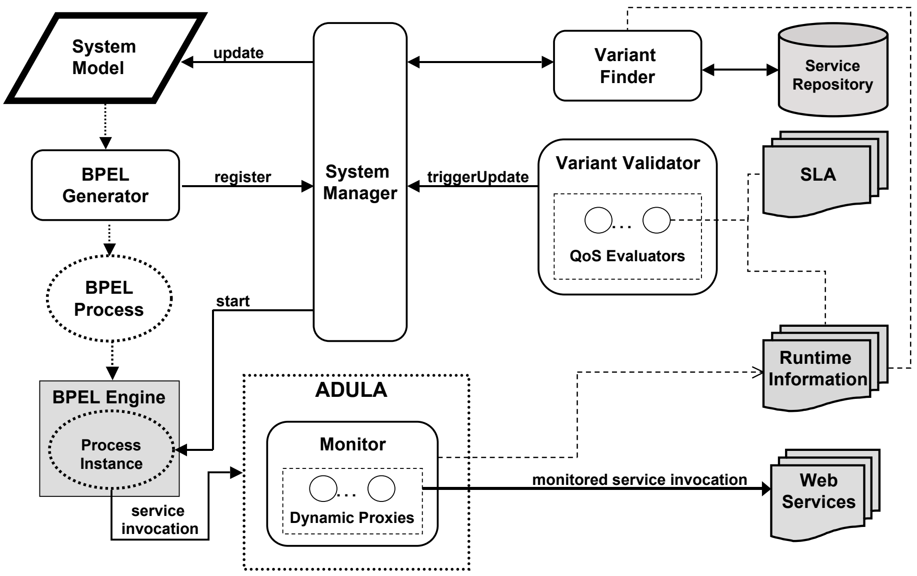

Fig. 3. Framework architecture

In Fig. 2 we show a fragment of the generated model for a smart home system. We detail the graph branch representing the temperature control functional requirement. The requirement is to adapt the temperature in the room according to weather conditions. The *Variant Finder* provides two variants for the temperature control variability point:

1. The system reads the weather forecast from an online service, gets the recommended temperature value for the external temperature forecast and sets the room heater to the recommended value.
2. The system reads the external temperature sensors installed on the building, computes the temperature value using a dedicated function and sets the room heater to the computed value.

3.2 Architecture

Fig. 3 presents the architecture of our framework. BPEL [7] is the de facto standard for service compositions. BPEL process definitions are deployed to BPEL engines that instantiate and execute a process instance when a request for the deployed process arrives. Our framework leverages the BPEL technology to implement the system.

The *System model* is the one introduced in Section 3.1. Our framework automatically generates BPEL processes from the system model. The processes are instrumented to allow monitoring the component services and the system QoS parameters. Based on monitoring information the framework estimates QoS values, periodically updating the model. The updated model is used to decide the runtime configuration.

The *BPEL Generator* creates a BPEL process for every variant in the model. All generated processes are registered to the *System Manager* (the arrow labeled **register** in the figure).

The *System Manager* is responsible with selecting the variants to be executed for fulfilling the system requirements. Some systems that are an aggregation of distributed applications, such as service-oriented systems, can be seen as a composition of functional requirements implemented independently. The system core is a dispatcher that forwards requests to the *System Manager*. We can automatically generate the dispatcher from the model. The developer can choose not to use the fully automated version, but only to forward requests for variability points execution to the *System Manager*.

The *System Manager* keeps a mapping between variants in the model and the corresponding BPEL processes. A variant is mapped to exactly one process. When the execution of a variability point is triggered, the *System Manager* checks the model for the variant to execute. The *System Manager* starts the variant execution by invoking the process corresponding to the selected variant (**start**). This approach allows the system to evolve with the system model, changes in the model being reflected in the system implementation.

Changes in the environment affect the system so that variants can become outdated, or new variants can be found. *Variant Finder* provides new possible variants based on runtime information, such as the availability of a new service. The *Variant Finder* updates the model with new variants, which can be then selected to be executed. In this way, new variants are easily integrated without disrupting the running system.

Our framework uses the monitoring capabilities of ADULA, a framework for fault tolerant execution of BPEL processes introduced in previous work [8,9]. The *Monitor* component of ADULA provides statistics on QoS parameters of services used by processes. Service invocations are redirected through *Dynamic Proxies* that measure the response time of the service. The *Monitor* observes the service execution, collects measurements and provides aggregated performance statistics.

Variant Validators test the fulfillment of the system quality requirements for each variant using statistic methods, such as Bayesian inference [10] or statistical hypothesis testing [11]. In previous work [9] we have used statistical methods to detect SLO violations for BPEL processes.

QoS Evaluators compute the estimated value of the QoS making use of monitoring information. Based on the estimated values, *Variant Validators* compute the deviation for each variant and each QoS parameter, and update the model with the new values (**triggerUpdate**). The deviation value is computed for each QoS using a provided function.

4 Related Work

KAMI [12] is a framework for model evolution by runtime parameter adaptation. KAMI focuses on Discreete Time Markov Chain [13] (DTMC) models that are used to reason about non-functional properties of the system. The authors adapt the non-functional properties of the model using bayesian estimations to update the parameters that influence the non-functional properties. The estimations are computed based on runtime information, and the updated model allows verification of requirements. Our framework focuses on using the model at runtime to improve a system's autonomic capabilities. We also consider new variants for functional requirements and use the evolved model to dynamically adapt the system.

Similarly to KAMI, the approach in [14] considers the non-functional properties of a system in a web-service environment. The authors provide a language, SLAng, that allows to specify QoS to be monitored.

There are different approaches to provide self-adaptive systems. Models@Run.Time [15] propose leveraging software models and extending the applicability of model-driven engineering techniques to the runtime environment to enhance systems with dynamic adapting capabilities. The system adaptation in our approach leverages this idea using the model to determine the system runtime configuration.

In [16], the authors use an architecture-based approach to support dynamic adaptation. Rainbow [17] also updates architectural models to detect inconsistencies and correct certain types of faults. In [18] the authors implement an architecture-based solution in the context of mobile applications to adapt the system by replacing the implementation of components at runtime. None of these solutions considers the impact of environmental changes on the quality requirements of the system.

A different approach to using models at runtime for system adaptation is taken in [19]. The authors update the model based on execution traces of the system. Our approach provides new execution paths for the system by integrating new and modified variants into the model.

The work in [20] provides a solution to a different issue concerning dynamically adaptive systems, which is the control over the wide number of variants that a system with many variability options can have. The authors introduce a solution to maintain dynamically adaptive systems by using aspects to evolve the model.

5 Conclusion

In this paper we outlined our approach to enhance a system's autonomic capabilities by using an automatically evolving model of the system at runtime. The model is periodically updated with re-evaluated QoS values based on runtime information. The evaluations can be used to predict and prevent QoS violations.

We gave an overview of a BPEL-based framework implementing our approach and introduced a system model leveraging service compositions. Variants represented in our flow graph model are implemented as BPEL processes that can be switched at runtime allowing the system to adapt to changes in the environment. In this way, the system can integrate new variants without requiring interruption of the running system.

Acknowledgements. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "SOSOA: Self-Organizing Service-Oriented Architectures" (SNF Sinergia Project No. CRSI22_127386/1).

References

1. Open Grid Forum: WS-Agreement specification, `http://www.ogf.org/documents/GFD.107.pdf`
2. Marconi, A., Pistore, M., Traverso, P.: Automated composition of web services: the astro approach. IEEE Data Eng. Bull. 31(3), 23–26 (2008)
3. Klusch, M., Fries, B., Sycara, K.P.: Owls-mx: A hybrid semantic web service matchmaker for owl-s services. J. Web Sem. 7(2), 121–133 (2009)
4. Constantinescu, I., Binder, W., Faltings, B.: Service composition with directories. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 163–177. Springer, Heidelberg (2006)
5. Lamsweerde, A.: Reasoning about alternative requirements options. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 380–397. Springer, Heidelberg (2009)
6. Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Chichester (2009)
7. BPEL: BPEL 2.0 standard specification, `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`
8. Mosincat, A., Binder, W.: Transparent Runtime Adaptability for BPEL Processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 241–255. Springer, Heidelberg (2008)
9. Mosincat, A., Binder, W.: Automated Performance Maintenance for Service Compositions. In: WSE, pp. 131–140 (2009)
10. Berger, J.: Statistical Decision Theory and Bayesian Analysis. Springer, Berlin (1999)
11. Romano, J.: Testing Statistical Hypotheses. Springer, Berlin (2005)
12. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: ICSE, pp. 111–121 (2009)
13. Meyn, S.P., Tweedie, R.L.: Markov Chains and Stochastic Stability. Springer, London (1993)
14. Raimondi, F., Skene, J., Emmerich, W.: Efficient online monitoring of web-service slas. In: SIGSOFT FSE, pp. 170–180 (2008)
15. Blair, G.S., Bencomo, N., France, R.B.: Models@ run.time. IEEE Computer 42(10), 22–27 (2009)
16. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
17. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)
18. Floch, J., Hallsteinsen, S.O., Stav, E., Eliassen, F., Lund, K., Gjørven, E.: Using architecture models for runtime adaptability. IEEE Software 23(2), 62–70 (2006)
19. Maoz, S.: Using model-based traces as runtime models. IEEE Computer 42(10), 28–36 (2009)
20. Morin, B., Barais, O., Nain, G., Jéźequel, J.M.: Taming dynamically adaptive systems using models and aspects. In: ICSE, pp. 122–132 (2009)