2010 14th IEEE International Enterprise Distributed Object Computing Conference

Runtime Adaptability through Automated Model Evolution

Adina Mosincat, Walter Binder, Mehdi Jazayeri
Faculty of Informatics
University of Lugano
Switzerland
{*adina.diana.mosincat, walter.binder, mehdi.jazayeri*}*@usi.ch*

***Abstract*—Dynamically adaptive systems propose adaptation by means of variants that are specified in the system model at design time and allow for a fixed set of different runtime configurations. However, in a dynamic environment, unanticipated changes may result in the inability of the system to meet its quality requirements. To allow the system to react to these changes we propose a solution for automatically evolving the system model by integrating new variants and periodically validating existing ones based on updated quality parameters. To illustrate our approach we present a BPEL based framework using a service composition model to represent the system functional requirements. Our framework estimates Quality of Service (QoS) values based on information provided by our monitoring mechanism, ensuring that changes in QoS are reflected in the system model. We show how the evolved model can be used at runtime to increase the system's autonomic capabilities and delivered QoS.**

***Keywords*-dynamic adaptability, model evolution, model at runtime, quality requirements**

I. Introduction

When reasoning about design decisions, software engineers must take into consideration two important concerns: firstly, runtime adaptability to allow the system to provide its functionality regardless of the changes in the environment, and secondly, system quality requirements, such as performance, reliability, and cost.

Autonomic systems must be able to continuously adapt to changes in the environment. Dynamically adaptive systems must be able to change behavior depending on environmental conditions and switch between runtime configurations without disrupting the running system. Such systems support adaptation by means of variants that determine runtime configurations and are specified in the system model at design time.

The first issue that engineers face is that the variants provided at runtime are fixed and non-exhaustive. In order to integrate new variants enabled by environmental changes or required by new user needs, the system must be modified and redeployed. The second issue the engineers face is that the delivered quality estimated at design time can be affected by parameter changes at runtime. These changes can result in the inability of the system to fulfill its quality requirements.

We propose a solution that addresses these issues in two phases:

1) We automatically evolve the system model by periodically updating it with monitored quality information and by integrating new variants enabled by changes such as the availability of a new service.
2) We use the updated model at runtime to select the system runtime configuration.

We present a BPEL-based working framework implementing the proposed solution. We use a service composition model to represent the running system which allows us to leverage the advantages of service oriented architectures: flexibility, loose coupling, and ease of integration.

The quality requirements of a service oriented system are expressed as Quality of Service (QoS) parameters guarantees, called service level objectives (SLO [28]), in service level agreements (SLA [28]). Given the distributed nature of service oriented systems, the system does not have control over the evolution of the composing services. Therefore, the QoS of the composing services can vary in time and might result in violations of the system SLOs.

We provide a monitoring mechanism that allows us to gather information on the running system and monitor the service execution. Based on monitoring information, our framework detects violations of the system SLOs and invalidates the violating variants. In this way, subsequent SLO violations are prevented and the system can maintain the required quality.

We update the system model with QoS values estimated based on monitoring information and use the updated QoS values to select the runtime configuration. By using the model at runtime, changes in the model are immediately reflected in the running system.

In this paper we propose a novel approach to automated model evolution for dynamically adaptive systems that (1) finds variants to adapt to changes in the environment and integrates them into the model and (2) annotates the model with updated values of delivered system quality computed based on monitoring information on the running system. Our framework enhances the system with autonomic capabilities by leveraging service compositions and using the automatically evolving model at runtime. This paper improves, refines and evaluates the approach outlined in a former position paper [27].

1541-7719/10 $26.00 © 2010 IEEE
DOI 10.1109/EDOC.2010.22

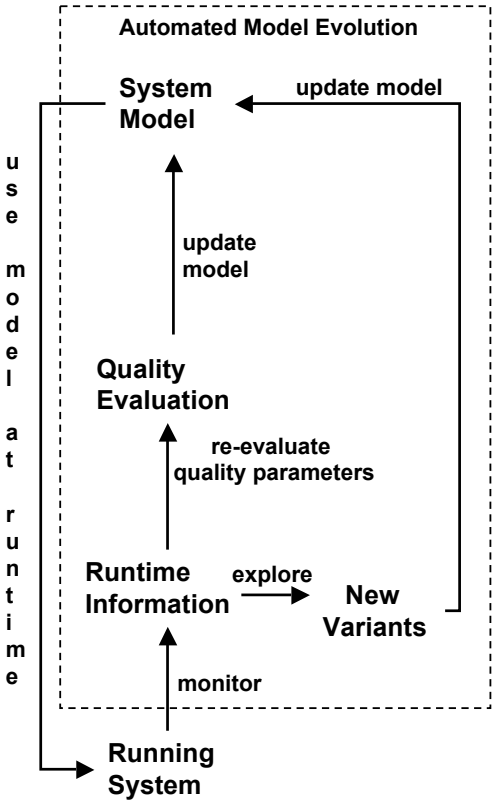

Figure 1. Overview of our approach to automated model evolution.

This paper is structured as follows: Section II introduces our approach and details our choice of model. Section III presents an overview of our framework, detailing the automated model evolution mechanism. In Section IV we describe the system adaptation mechanism discussing the framework interactions for variant selection, and we evaluate our framework in Section V. We discuss related work in Section VI. Section VII concludes this paper.

II. Overview and System Model

In this section we present a general overview of our approach and describe the system model we use to represent the running system.

A. 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.

Figure 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.

We take into considerations two types of changes: changes that might affect the delivered quality of the system and changes that might determine structural modifications of the model, such as adding a new variant.

At design time the delivered quality values are estimated using available parameters, such as QoS guarantees specified by SLAs, or provided by domain experts. However, these parameters might change because of system evolution or changes in the environment. Thus, the initially estimated delivered quality values become obsolete and the changes might result in violations of the system quality requirements. To avoid these situations, the quality estimations must be periodically updated to reflect the current state of the environment.

One important part in our approach is gathering information on the current environmental conditions by monitoring the running system. The gathered information is then used (a) to re-estimate the delivered quality estimations and (b) to detect the system's inability to fulfill its quality requirements. In the Quality Evaluation step, the delivered quality values are re-estimated based on the monitoring information. The model is then updated according to the new estimations.

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 framework can improve the system delivered quality and prevent violations of quality requirements by leveraging the estimations when selecting the variant to execute.

B. Model

The choice of the system model plays an important part in achieving the required autonomic capabilities. In the following we describe our system model based on service compositions. We take into consideration goal oriented models, such as [31], [17].

The model we describe below was introduced in [27]. Figure 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 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.

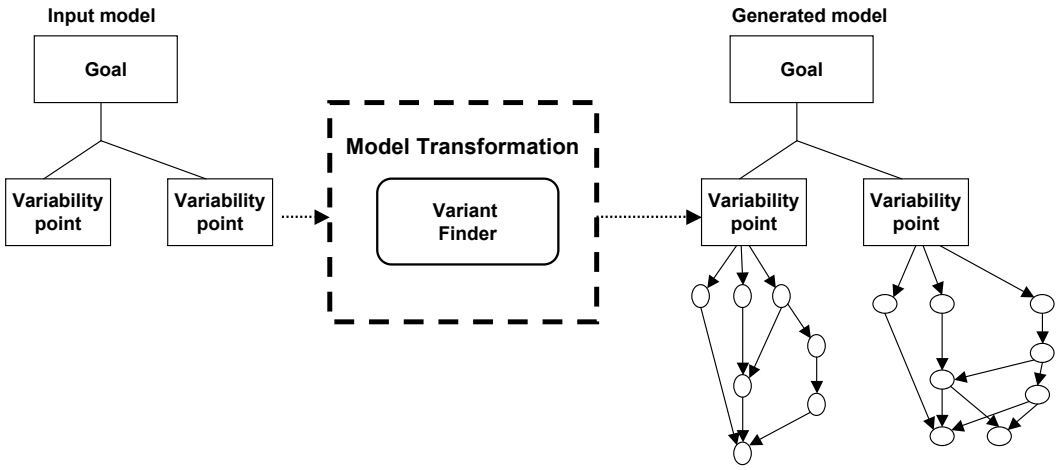

Figure 2. Model transformation.

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 [21], [16]. The variability points must be expressed in a query language understood by the Variant Finder [2], [10], and the services must be semantically annotated.

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 [17]. A goal model is an AND/OR graph showing how higher-level goals are satisfied by lower-level ones (goal refinement) [18]. 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 pair <Variant ID, {<qosID, value, flag>} > in which:

1) Variant ID uniquely identifies the variant.
2) The set contains the values of the estimated QoS values for the variant. At model generation, the QoS values are computed based on the SLAs of the variant's composing services. The QoS parameter is identified by qosID. A value of FAILED for the flag invalidates the variant.

In Figure 3 we show a fragment of the generated model for an astronomical observatory system. We detail the graph branch representing the weather forecast functional requirement. The requirement is to provide an image of a given targeted area predicting the weather conditions for the next 24 hours. The Variant Finder provides three variants for the weather forecast variability point:

1) The system gets a satellite image from the dedicated satellite for the targeted area and computes the weather prediction based on information such as cloud type and wind strength and direction. The predictions are rendered in an weather forecast image.
2) The system gets up to four images from the satellite network for the areas closest to the targeted area. It then matches previously acquired images from the areas, crops the image parts that cover the targeted area and renders a satellite image of the targeted area by putting together the cropped parts. It then uses the obtained image to compute the weather prediction and render the weather forecast image.
3) The system uses a cached image of the targeted area and computes the deviation of the weather conditions from the ones computed for the cached image based on the weather changes observed between the taken time of the cached image and the current time. It then renders the weather forecast image.

III. Framework Architecture.

This section presents the architecture of our framework that illustrates the approach introduced above and describes

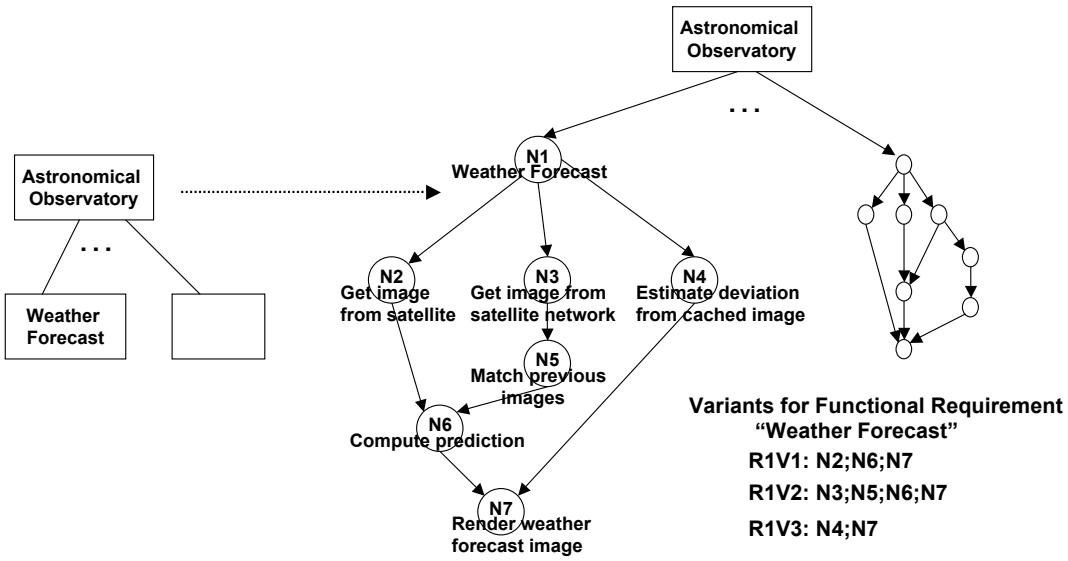

Figure 3. Astronomical observatory system: weather forecast functionality example. There are three variants for the weather functionality, identified by R1V1, R1V2, respectively, R1V3.

the components of the framework that play an important part in the automated model evolution and system adaptation.

A. Overview

Figure 4 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 II-B. 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* has the following responsibilities:

1) selects the variants to be executed for fulfilling the system requirements.
2) keeps a mapping between variants in the model and the corresponding BPEL processes. A variant is mapped to exactly one process.
3) updates the system model (the arrow labeled **updateModel** in the figure).
4) manages variant identifiers (variantID).

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.

When the execution of a variability point is triggered, the System Manager checks the model for the variant to execute. The variant is selected according to criteria implemented by Selection Policies. 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. In our current implementation we assume new variants are manually provided. Our framework allows for integration of more complex systems for automatic variant discovery. One such system is the matchmaker described in [16] that takes the requirement description and queries the service repository finding the set of service compositions that solve the requirement.

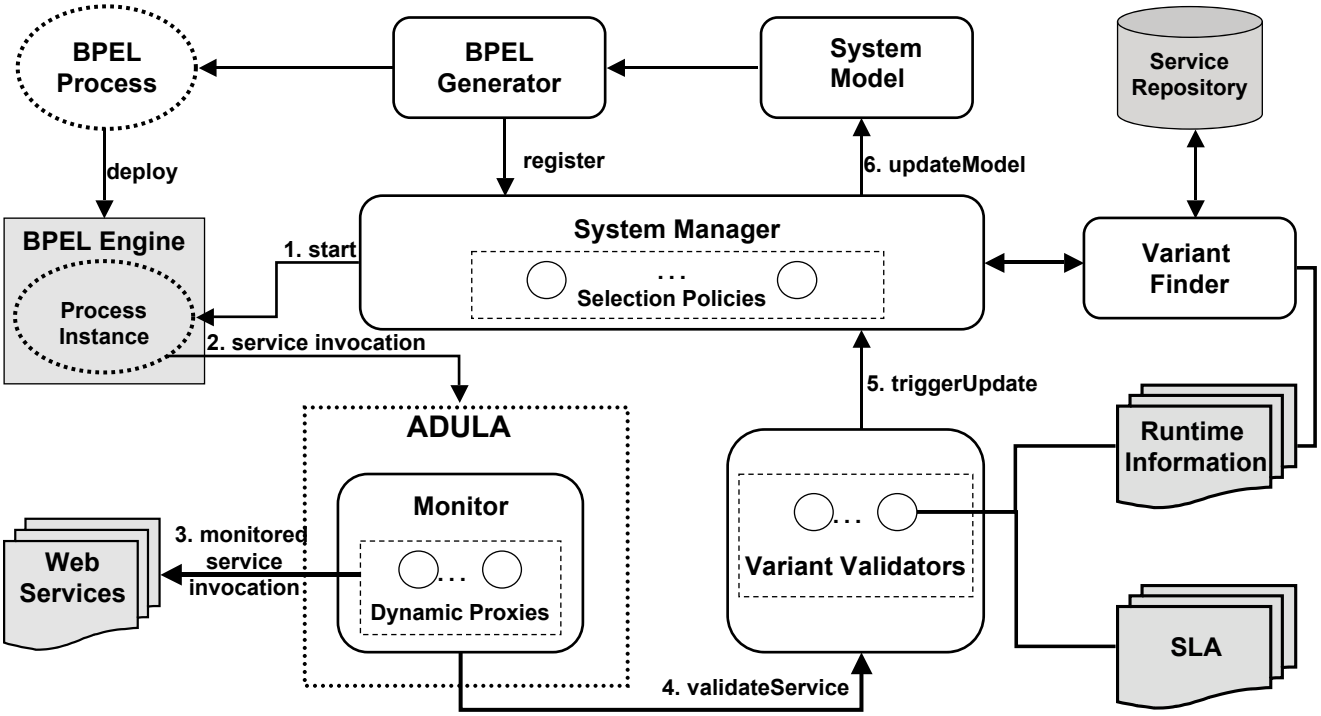

Figure 4. Framework architecture. The grey boxes represent external components and data. The numbered arrows represent the flow for a variant execution.

Usually service discovery and matchmaking is a time consuming process. The search for a new variant solving a variability point is triggered upon availability of runtime information signaling a change, or upon request from the System Manager. Upon the invalidation (violation detection) of a variant, the System Manager checks the availability of existing variants for the corresponding variability point. If no valid variants exist for the variability point or the QoS values are not satisfactory (determined by checking them against a given threshold), the System Manager requests the Variant Finder for new variants solving the variability point.

Our framework uses the monitoring capabilities of ADULA, a framework for fault tolerant execution of BPEL processes introduced in previous work [25], [26]. 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 [5] or statistical hypothesis testing [33]. In previous work [26] we have used statistical methods to detect SLO violations for BPEL processes.

B. Variant Validation

Variant Validators compute the estimated value of the QoS making use of monitoring information. Based on the estimated values, Variant Validators validate each variant for each QoS parameter, and update the model with the new values (**triggerUpdate**). In case a violation is detected, i.e., the variant does not fulfill the system quality requirement, the validator invalidates the variant by setting the flag corresponding to the QoS parameter on *FAILED*.

Below we give two examples of possible Variant Validators. A simple Variant Validator computes the system QoS parameters using a provided utility function specific to each QoS parameter [19]. For instance, for response time the variant QoS values is computed as the sum of the QoS parameters of the individual services used by the variant. Initially, the validator uses as QoS parameter source the SLAs of the services. The validator uses monitoring information when it becomes available.

As another example, for SLAs that require the average (arithmethic mean) QoS value not to exceed a given threshold, we apply the Student t-test statistical significance test [33] to determine the probability of a QoS parameter to be violated. Let's consider as an example that the SLA guarantees that the system average response time does not exceed 2500ms. In this case we make use of our monitoring mechanism that provides statistics on services response times and variant execution times. The null-hypothesis is that the requirement is not violated, i.e., we assume that the average process response time is smaller or equal to the given threshold, in this case, $\leq$ 2500ms. The statistical test tells us whether the samples, i.e., the measured response times for a variant, are unlikely to have occurred by chance given the truth of the null-hypothesis. In that case, the null-

hypothesis is rejected, and the validator signals a violation.

The new estimated QoS value is the sample mean computed from the available samples. The significance level α, the probability of committing a type-I error, i.e., the probability of rejecting the null-hypothesis when it is true, is a configurable value. When multiple variants exist and the tolerance for deviations from quality requirements is critically low, the α values may be unusually high (e.g., $\alpha = 0.1$). High values for α mean that there might be a larger number of false positives (the validator detects a false violation), on the other hand they assure a faster reaction of the system to the degradation of the delivered quality.

This technique is suited for variants (functionalities) that are often executed and for which enough monitoring information can be gathered, i.e., a sufficient number of samples exists. Variant Validators using null-hypothesis statistical significance tests need access to some history of previous samples (i.e., variant execution times), and as a consequence, for variants for which little monitoring data is available, SLA violations may be detected with some delay.

We verify that the variant execution times (the samples) follow a normal distribution using the Chi-Square normality test [9]. In case the Chi-Square test is negative, i.e., the distribution is not normal, we use non-parametric tests, such as the Wilcoxon signed-rank test [35] for one sample.

IV. System adaptation

In this section we describe system adaptation enabled by our framework detailing how variants are selected at runtime.

To provide a functionality, the System Manager selects a variant from the model and executes it by starting an instance of the process corresponding to the selected variant. Selection Policies implement criteria for selecting the variant to be executed for each variability point. Selection Policies are customizable and are mapped to variability points. For variability points for which no custom Selection Policy is provided, the default Selection Policy is used. Selection Policies allow for prioritization of QoS parameters.. For instance, the selection policies prioritizing cost can determine the selection of a variant that violates the performance requirement but obtains a better cost, or other policies can specify that the variant with best performance should be selected regardless of the value of the other QoS parameters. The policy can also prioritize the variants by specifying preferred variants. This feature is useful when alternative ways of implementing the functionality should be used only in case of failure. 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 if the alarm device is out of order.

The default Selection Policy selects the variant that offers the best QoS parameter values. Figure 5 shows the peudo-code of the default policy. The policy assigns a ranking

```
forall qos do
begin
    forall variants do
    begin
        if flag_qosID =FAILED then
            negativeRanking_variantID ←
                    negativeRanking_variantID + 1
        else
            add(qosValues, <variantID, value>)
        end
    end
    sortedValues←sort(qosID, qosValues);
    forall variantID ∈ sortedValues do
    begin
        positiveRanking_variantID ←
            positiveRanking_variantID +
                position(sortedValues, variantID)
    end
end
forall variants do
begin
    if negativeRanking_variantID = min negativeRankings
    then
        if positiveRanking_variantID = max positiveRankings
        then
            selected ← variantID
        end
    end
end
```

Figure 5. Selection Policy example pseudo-code. The policy ranks the variants according to their degree of fulfillment of the system QoS requirements and selects the best available variant.

to each variant for every available QoS parameter. For variants that violate a system SLO for a QoS parameter, the policy assigns to the variants a negative ranking of 1. For every QoS parameter, the policy computes a ranking list (`sortedValues`) that ranks all non-violating variants from the worst (e.g., for response time parameter the variant with the longest response time) to the best (e.g., for cost parameter, the variant with the lowest cost). The policy computes for every variant a negative ranking sum that represents the number of violations and a positive ranking sum that represents the sum of positions in the ranking lists. The variant with the lowest negative ranking sum and the highest positive ranking sum is selected. The function `sort` creates a ranking list for the QoS parameter values `qosValues` according to an utility function defined for the QoS parameter identified by `qosID`. The function `position` returns the index of the `variantID` in the `sortedValues` ranking list.

Figure 6 represents the variant selection and framework interactions in case of an SLO violation. For clarity, only the important functionalities are presented. `getVariants(pointID)` parses the system model and reads the variants along with the data contained by the variability point node identified by `pointID`. The data is made available to the selection policy through the variants identifiers. `selectVariant(Set{variantID})` selects the variant to be executed based on the model data and selection criteria implemented by the Selection Policy.

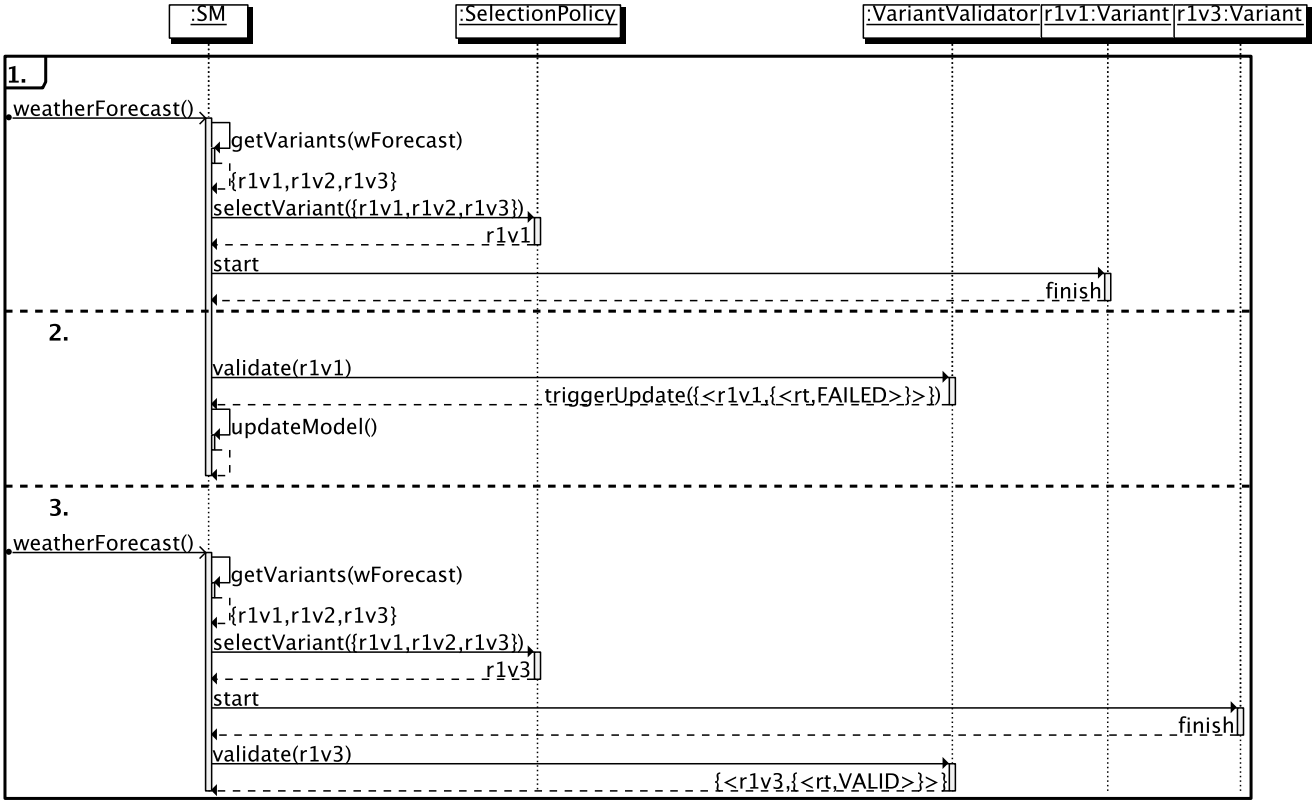

Figure 6. Variant selection for weather forecast functionality.

1) First functionality request
   When the request for the weather forecast functionality arrives, the system selects variant *r1v1* and executes it.
2) SLO violation
   Upon the variant finish, the *Variant Validator* computes the QoS parameter values and detects a violation of the response time SLO. The variant is invalidated by setting the response time parameter flag to *FAILED* and updating the model with the computed QoS values.
3) Second functionality request
   Upon sub-sequent requests for the weather forecast functionality, the System Manager selects a valid variant from the updated model and executes it (*r1v3*). The Variant Validator periodically checks the invalidated variants to determine if services have recovered.

V. Evaluation

In this section, we evaluate our framework by exploring the response time of the system using our framework when the response time of composing services degrades. We compare different validation techniques exploring the effectiveness of the statistical tests for violation detection.

Our evaluation is based on the astronomical observatory system example. For the implementation of the weather forecast functionality, the framework provides one BPEL process for each of the variants. The first variant interacts with three services: one to get the satellite image of the area from the plant's satellite, one to compute the weather forecast based on the satellite image (we will call this service *forecaster*), and one to render the computed forecast image (*renderer*). The second variant interacts with four services: one to get a number of satellite images from the network, one to extract the satellite image of the area from the intersection of all the satellite images, the *forecaster* service and the *renderer*. The third variant interacts with two services: one to extract the satellite image of the area by estimating modifications based on previously cached images and compute the forecast image, and the *renderer*.

We take into consideration the response time parameter, and consider that the astronomical observatory system guarantees through its SLA that the weather forecast functionality is provided with an average response time less than 2500ms. The selection policy defines the first variant as the preferred one, meaning that the system executes the first variant whenever possible switching to the best available of the other two variants only in case the first variant violates the system QoS requirements.

We developed a testbed which models web service performance with discrete time Markov chains [22]. The testbed includes web services, client workload generators, as well as performance measurement tools.

For measuring response time, we use for the *forecaster* service a service model with five states, in each state the service has a different response time. In the state fast the response time of the service is 500ms, in the state slow, the response time is 2500ms, respectively the response times in the intermediary states are 1000ms, 1500ms, and 2000ms. The *forecaster* service changes state on every time slot[1] until it reaches the slowest state (2500ms) and afterwards remains in the same state. All other services are in the fast state (500ms) and do not change state. The transition period is 30 seconds.

Our implementation uses Java 5, Apache Axis 1.4, and BPEL 2.0; as BPEL engine we use ActiveBPEL 4 [1]. Both ADULA and the BPEL engine are deployed in an Apache Tomcat 4.1.24 installation. The variant validators using statistical tests are implemented with the Apache Commons Mathematics 2.0 library. Our measurement machine is an Intel Core 2 Duo (2.4GHz, 2GB RAM) running Mac OS X v10.4. All measurements were repeated 15 times and we report the median of these measurements.

We compare the response time of the system when using a simple Variant Validator, respectively when using a Variant Validator implementing the Student t-test with low and high α values.

Figure 7 shows the response time of the system for 100 variant executions[2]. The simple Variant Validator detects a violation as soon as the variant response time exceeds 2500ms, invalidating the first variant. The system switches to the best available non-violating variant (the third one in this case) so that the system response time requirement is fulfilled. The Variant Validator using the Student t-test tests that the average response time of the system is below 2500ms. When using a high α value (e.g. 0.1) the violation is detected sooner, but there is a higher risk of false positives. The simple validator detects the violation very fast, allowing the system to obtain the best performance. On the other hand, it can result in a larger number of false violations than the validator using the Student t-test. However, as the validator periodically checks the recovery of the invalidated variants, the impact of false violations is reduced.

Figure 8 shows the average response time of the system after every completed 15 variant executions. The overall average response time of the system is 2215.8 ms when using the Student t-test with α=0.0005, 2071.6 ms when using the Student t-test with α=0.1, respectively 1800.7 ms when using the simple validator. The system response time requirements is maintained in all cases.

In summary, our framework successfully detects SLO violations preventing the degradation of the delivered quality. The best suited technique depends on the system. For systems that specify variants that are equal from the user preference point of view, using the simple validator obtains the best delivered quality. On the other hand, for systems that specify preferred variants, using statistical techniques allows for a more accurate violation detection.

[1]A *time slot* is the moment in time when a decision is made randomly based on the current state and the transition probability to change or to keep state.

[2]A variant execution represents the execution of a process instance.

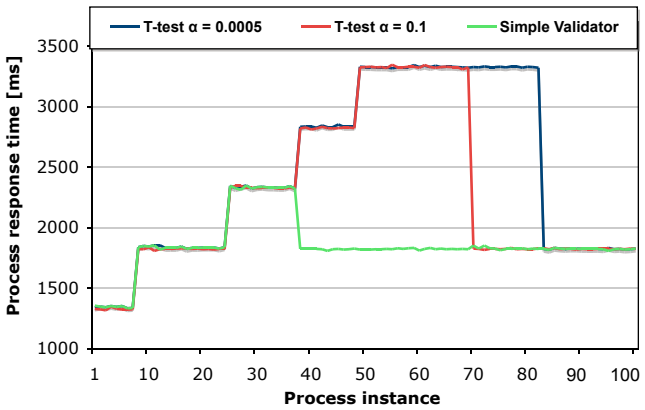

Figure 7. System response time using the Student t-test validator with α=0.1, α=0.0005, respectively, the Simple Validator.

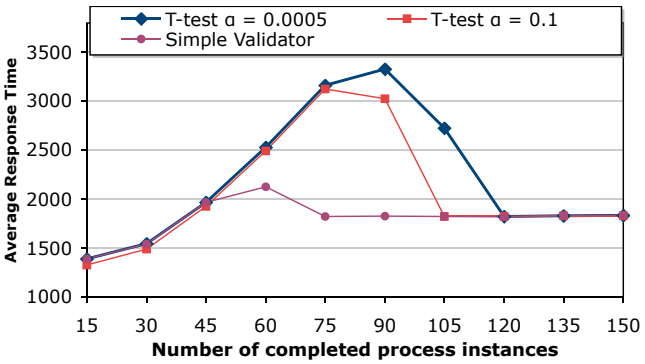

Figure 8. Average system response time for every 15 variant executions, using the Student t-test validator with α=0.1, α=0.0005, respectively, the Simple Validator.

VI. Related Work

Existing solutions that take into consideration non-functional properties of a system in model evolution, such as [11], [32], do not use the evolved model at runtime to increase the system's autonomic capabilities.

KAMI [11] is a framework for model evolution by runtime parameter adaptation. KAMI focuses on Discreete Time Markov Chain [22] (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 [32] 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.

CEA-Frame [34] is a framework for system adaptation depending on context that combines aspect oriented techniques with model driven engineering. The framework separates the QoS concerns from the application and provides QoS-aware planning and adaptation to suit the operation context and the resource availability. Our approach also considers evolving the model based on observed quality values using statistical techniques.

The approach taken in [30] also combines aspect oriented techniques with model driven engineering to weave QoS concerned aspects within the system model in order to allow for monitoring of the QoS parameters.

There are different approaches to provide self-adaptive systems. Models@Run.Time [6] 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 [29], the authors use an architecture-based approach to support dynamic adaptation. Rainbow [15] also updates architectural models to detect inconsistencies and correct certain types of faults. In [14] 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.

In a different context, in [8] the authors introduce a technique of enhancing a system's autonomic capabilities by using variability models at runtime. They focus on mass-production environments that provide systems which need to change behavior depending on user context, but offer limited customization options, and argue that autonomic capabilities can be achieved by reuse of variability models. Our approach allows variants to be added at runtime, not requiring them to be provided at design time.

A different approach to using models at runtime for system adaptation is taken in [20]. 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 [23] 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.

In the world of service compositions, ALBERT [3] is a language that is used to specify functional and non-functional property assertions which are verified at design time by model checking and used at runtime as dynamically evaluated assertions. Our approach takes dynamic models into consideration, i.e., models that can evolve at runtime because of environmental changes.

There are a number of solutions that allow monitoring of the composing service execution and provide runtime adaptability of service compositions by means of dynamic binding [24], [4], [13], [12], which could be used as external runtime information providers. They do not take into consideration model evolution, nor QoS estimations updates.

VII. Conclusion and Future Work

In this paper we introduced a novel approach to enhancing 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.

Our framework demonstrates how service compositions can be leveraged to achieve dynamically adaptive systems. 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.

The model we leverage inherits the limitations of automated service composition, because it requires variants to be expressed as a simple flow of service operations. We are exploring possibilities to optimize the framework by using more complex BPEL constructs when generating the variant implementation.

As future work, we are considering different techniques for estimating the QoS parameters and predicting quality degradation, to determine which techniques are more effective for different types of systems.

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] Active Endpoints. ActiveBPEL engine. http://www.activevos.com/.

[2] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Semantic web query languages. In *Encyclopedia of Database Systems*, pages 2583–2586. 2009.

[3] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, 2007.

[4] L. Baresi, C. Ghezzi, and S. Guinea. Towards Self-healing Composition of Services. In *Contributions to Ubiquitous Computing*, pages 27–46. Springer Berlin / Heidelberg, 2007.

[5] J. Berger. *Statistical Decision Theory and Bayesian Analysis*. Berlin: Springer-Verlag, 1999.

[6] G. S. Blair, N. Bencomo, and R. B. France. Models@ run.time. *IEEE Computer*, 42(10):22–27, 2009.

[7] BPEL. BPEL 2.0 standard specification. http://docs.oasis-open.org/ wsbpel/2.0/wsbpel-v2.0.pdf.

[8] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *IEEE Computer*, 42(10):37–43, 2009.

[9] H. Chernoff and E. L. Lehmann. The Use of Maximum Likelihood Estimates in χ^2 Tests for Goodness of Fit. *The Annals of Mathematical Statistics*, 25(3):579–586, 1954.

[10] L. O. B. da Silva Santos, G. Guizzardi, L. F. Pires, and M. van Sinderen. From user goals to service discovery and composition. In *ER Workshops*, pages 265–274, 2009.

[11] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE*, pages 111–121, 2009.

[12] A. Erradi, V. Tosic, and P. Maheshwari. MASC-.NET-Based Middleware for Adaptive Composite Web Services. In *ICWS '07: Proceedings of the IEEE International Conference on Web Services*, pages 727–734, 2007.

[13] O. Ezenwoye and S. M. Sadjadi. TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services. In *WEBIST-2007.International Conference on Web Information Systems and Technologies*, 2007.

[14] J. Floch, S. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[15] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[16] M. Klusch, B. Fries, and K. P. Sycara. Owls-mx: A hybrid semantic web service matchmaker for owl-s services. *J. Web Sem.*, 7(2):121–133, 2009.

[17] A. Lamsweerde. Reasoning about alternative requirements options. In *Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos*, pages 380–397, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] A. Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley, 2009.

[19] N. B. Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, and V. Issarny. Qos-aware service composition in dynamic service oriented environments. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[20] S. Maoz. Using model-based traces as runtime models. *IEEE Computer*, 42(10):28–36, 2009.

[21] A. Marconi, M. Pistore, and P. Traverso. Automated composition of web services: the astro approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

[22] S. P. Meyn and R. L. Tweedie. *Markov Chains and Stochastic Stability.* London: Springer-Verlag, 1993.

[23] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132, 2009.

[24] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 815–824, New York, NY, USA, 2008. ACM.

[25] A. Mosincat and W. Binder. Transparent Runtime Adaptability for BPEL Processes. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 241–255, 2008.

[26] A. Mosincat and W. Binder. Automated Performance Maintenance for Service Compositions. In *WSE '09: The 11th IEEE International Symposium on Web Systems Evolution*, pages 131–140, 2009.

[27] A. Mosincat, W. Binder, and M. Jazayeri. Dynamically Adaptive Systems through Automated Model Evolution using Service Compositions. In *SC '10: Proceedings of the Software Composition Conference*, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Open Grid Forum. WS-Agreement specification. http://www.ogf.org/ documents/GFD.107.pdf.

[29] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[30] G. Ortiz and B. Bordbar. Aspect-oriented quality of service for web services: A model-driven approach. In *ICWS '09: Proceedings of the IEEE International Conference on Web Services*, pages 559–566, 2009.

[31] D. A. C. Quartel, W. Engelsman, H. Jonkers, and M. van Sinderen. A goal-oriented requirements modelling language for enterprise architecture. In *EDOC*, pages 3–13, 2009.

[32] F. Raimondi, J. Skene, and W. Emmerich. Efficient Online Monitoring of Web-service SLAs. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 170–180, New York, NY, USA, 2008. ACM.

[33] J. Romano. *Testing Statistical Hypotheses.* Berlin: Springer-Verlag, 2005.

[34] A. Solberg, D. M. Simmonds, R. Reddy, R. B. France, S. Ghosh, and J. Ø. Aagedal. Developing distributed services using an aspect oriented model driven framework. *Int. J. Cooperative Inf. Syst.*, 15(4):535–564, 2006.

[35] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1:80–83, 1945.