Towards System Viewpoints to Specify Adaptation Models at Runtime

Steffen Becker
Heinz Nixdorf Institute & University of Paderborn
steffen.becker@upb.de

September 14, 2011

Abstract

Current research focuses on the specification, analysis, and realisation of systems with self-* properties. Such systems are characterised by the fact that they react on changes in their environment by altering their own structure in order to maintain their non-functional properties. However, there is no standard engineering method to model such systems. In this paper, we propose a set of new system viewpoints and their views. Additionally we describe initial ideas of the properties we want to check based on the new system views.

1 Introduction

Today, many software systems have to react on changing environments to maintain quality of service properties as specified in their requirements documents. In the context of this paper, let us consider a publicly available service in the Internet which is required to be elastic, i.e., adapt itself to the number of concurrently executed requests by acquiring and releasing computing resources from a cloud [**?**] service, like Amazon's cloud or Microsoft's Azure. In order to engineer such a system in a model-driven way, engineers need to create a model of the system including the system's adaptation mechanism.

However, today's standard modelling language, UML2.3 [**?**], does not sufficiently support specifying adaptation mechanisms thus making it difficult to model self-adaptive systems out-of-the-box in UML. Also no other standard languages for this task exist at the moment due to the novelty of most of the involved techniques (Cloud computing, runtime adaptions, models at runtime). Thus, we need a metamodel to describe the runtime adaptation behaviour.

In this paper, we present initial requirements of such a model based on the example introduced above. The requirements lead to the definition of two new viewpoints and their views. This is a necessary prerequisite to define a model-driven approach to adaptive systems. Based on the newly described viewpoints, we argue on the advantages that system engineers gain when they start modelling their systems using these viewpoints. The new viewpoints were inspired by work in the area of component-based embedded software development using Mechatronic UML [**?**] and enhanced for the cloud computing domain.

The contribution of this paper is a proposal of a set of views to enrich existing languages like the UML2 or the PCM [**?**]. More specifically, we focus on the specification of self-adaptive component- or service-based systems with elasticity requirements, i.e., systems that can scale up or scale down during runtime. Based on the proposed views the paper briefly discusses the properties we would like to analyse based on the specifications and an implementation framework to realise such adaptive systems.

2 Adaptation Viewpoints

Modelling Requirements The main characteristic of a self-adaptive system is that it changes its runtime state to adapt to changes in its environment. However, the runtime state of a system is a broad concept ranging from the instances of the classes, components, etc. to the values of variables or the entries in the rows of an attached database. Therefore, we need to focus to enable a realistic specification.

For component-based systems, we consider *instances of the component types*, *instances of component connectors* and, if available in the enhanced component-meta model, *instances of the component ports* as the most important runtime aspects for self-adaptation as they change over time. To illustrate these elements, consider the example presented in Figure 1.

Figure 1 exemplifies a simple server side setup for an elastic system. It contains a load balancer which distributes incoming requests to the least loaded server. Server processes run on separate machines (today ideally hosted in a cloud). Cloud servers can be acquired and released as needed.

Such a system contains three important aspects that we reflect in our modelling viewpoints. First, the system has a *static type-level* aspect describing the general interaction of the system's elements. Second, such systems need to collect monitoring data (e.g. user arrival rates) from their environment to reason on their context. Hence, the system has a *monitoring*

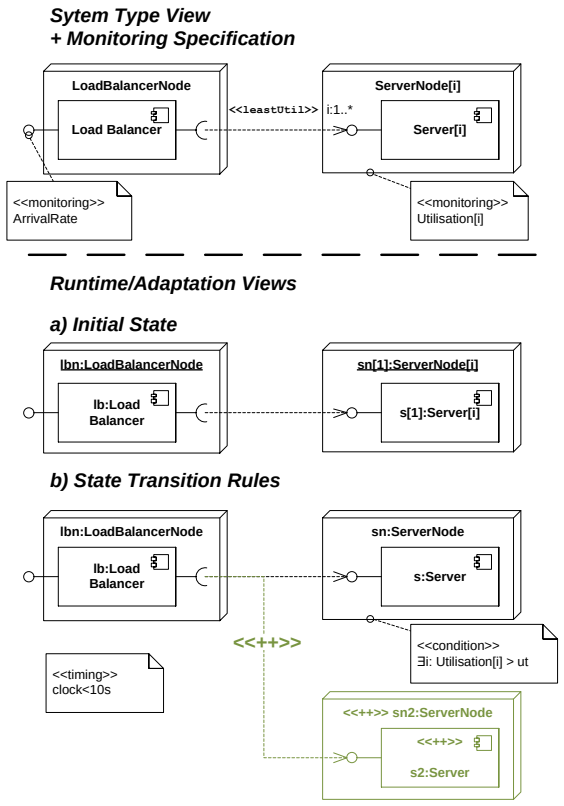

Figure 1: System views needed to describe self-adaptation

aspect to indicate the metrics (e.g. CPU utilisation, passage times, etc.) collected at runtime to create a context model. Third, the system needs a set of *rules and their preconditions* describing their reaction to changes in their environment.

To model the introduced aspects, we represent a system under two introduced viewpoints: a) the *system type viewpoint* combining a static view, a monitoring specification view and an allocation view of the system and b) the *runtime viewpoint* containing a view for the system's initial runtime state and views for its runtime adaptations. We explain each of these views in the following starting with the views from the type viewpoint.

System Type Viewpoint The system type viewpoint (Figure 1, upper part) shows adaptation invariant views of the system. It shows that the load balancer communicates at runtime with 1..* servers. Notice the extended labels in the static system model which allow to indicate which elements occur multiple times at runtime (indicated by an index variable i which is associated to the 1..* connector end and used to identify single server and component instances). Also invariant to the system runtime is the type of connector. In our case the connector routes calls to the least utilised server. Notice that the use of the stereotype `<<leastUtil>>`selects a connector from a predefined set of connectors which need to exists in a library of connectors if automated transformations should process the model in later development phases.

To allow this, the system also needs a type level *monitoring* specification. It tells the system which runtime monitoring probes need to be instantiated and applied to the system to instrument it. In our example, we want to monitor the overall rate of requests arriving at the load balancer and the utilisation of each of the servers. The type level specification already defines the variable $Utilisation[i]$ to store the measurement of the ith server's utilisation. There are a number of metamodels or languages for application servers to specify such monitoring probes which can be reused here. One example among many others is the Probe Specification language which is included in the PCM to specify measurement probes for the PCM's simulation.

Notice finally, that we did not talk about the system's behaviour yet. On the system's type level, we are only allowed to use component types for which we assume an existing behaviour description. As long as these components are stateless (as in our example) this is sufficient for the runtime viewpoint as presented next. In case of stateful components things get more complex as the runtime viewpoint then needs to include even more views.

Runtime Viewpoint The runtime viewpoint describes the changing parts of the system at runtime while the system adapts to its environment. It should not replicate any of the information already present in the system type level but always refer to it. From our example system's type specification we can derive that the only model elements which can change at runtime in adaptation actions are the active server nodes and their server component instances. In addition, we have a model storing the measurements of the specified monitoring probes (not depicted in Figure 1). These two models are sufficient to maintain our minimal *model at runtime*. However, for presentation purposes, Figure 1 (bottom part) repeats also invariant model parts like the single load balancer instance.

The runtime viewpoint consists of two views: the initial state view and the state transition views.

The initial state view specifies the instance of the system type model which reflects the system's runtime state at the beginning of the system's execution. In our example system, the view models the fact, that initially we start with a single server attached to our load balancer component.

The state transition view specifies how the system changes its model at runtime. The system itself is then changed accordingly to resynchronise the system's model with the system's realisation. As an initial modelling approach, we plan to use timed graph transformation systems [**?**]. Such transition systems allow the specification of the change of typed graph

structures to reflect the changes of the model at runtime. For example, the rule given in Figure 1 specifies the addition of a new instance of a server to our system in case of overloads. New elements are added to our model at runtime as indicated by the green model elements marked with a `<<++>>` stereotype. The rule additionally contains a `<<timing>>` stereotype which describes the upper time bound of the reconfiguration process, i.e., the time needed to change the real system's state to be synchronized with the model again. Additionally, we plan to include guarding preconditions for the rules. While the rules themselves describe the change of the instance of the system's type model over time, the guards (indicated by the `<<condition>>` stereotype) query the monitoring model at runtime in order to reflect the context dependent reactions of the system. In our example, an adaption is initiated if any of the servers reaches the specified utilisation threshold.

Using the Introduced Viewpoints Any model is useless without specifying its pragmatics, i.e., the purpose for which you can use it. In our case, several opportunities arise from the new modelling viewpoints.

First, we can perform consistency checking between the views of the system. For example, we could verify that the initial system state is an instance of the system's type view and that, by applying the adaptation rules, we never reach a state of the system not complying any longer to the system's type viewpoint.

Second, we can use the new viewpoints to analyse the quality properties of the system to check whether the system complies to its adaptation requirements (in case the enhanced model already allowed quality predictions, e.g. the PCM). For example, we could demand our system to respond in 99% of all situations where it is not adapting in less than 1 sec. In cases when it is adapting, it should respond in 90% of all cases in less than 1sec. Additionally, we could check that the system is cost and energy efficient because of its elasticity (ability to scale up and down) by checking that it always consumes a minimum of virtual servers from the cloud.

Finally, we can implement a model-transformation transforming our system's specification into platform dependent implementations based on some framework for managing virtual cloud servers and load balancing techniques.

3 Related Work

Most of the underlying ideas presented in this paper are not new in literature. However, our contribution is the combination of existing techniques and their application in the cloud computing domain to model, analyse and manage self-adaptive, elastic cloud applications. The idea as presented in this paper was inspired by graph transformation systems [**?**], timed graph transformations [**?**] and their application in the embedded system's domain to specify self-coordinating systems using MechatronicUML [**?**]. In this domain, the graph transformation system is combined with model checking techniques to ensure that the system never reconfigures itself such that it reaches an unsafe state.

In the ADL community, Batista et al. [**?**] identified three viewpoints for self-adaptive systems: the style viewpoint, the instance viewpoint and the runtime viewpoint. We did not consider an explicit style viewpoint in this paper, but the other two match to this paper's understanding. However, Batista et al. extended ADLs and use their language mainly for implementation while this paper considered UML like system models and focuses on using the models at runtime for predictions.

Bencomo et al. [**?**] propose the Genie framework to model, generate and operate self-adaptive component-based systems. For the transition rules they choose to take explicit variability options instead of a graph transformation approach. They also do not consider the design time analysis of the system.

Ardagna et al. [**?**] propose a software quality model-driven framework embedding the needed information for models at runtime into the typical PIM and PSM terminology by the MDA standard. However, they do not elaborate on the specification of viewpoints for modelling but concentrate on quality analysis models.

4 Conclusions

This paper proposes new viewpoints and their views to support the modelling of self-adaptive systems. The viewpoints translate ideas from embedded system modelling in MechatronicUML into the context of elastic cloud computing applications.

The new viewpoints are expected to help in specification, analysis and implementation of self-adaptive systems. They are a prerequisite for a model-driven approach towards self-adaptive systems and rely on models at runtime. Software architects and system quality analysts should use them to engineer self-adaptive elastic systems.

Future work needs to define the meta-models of the introduced views, define transformations into implementations and analysis models. Existing tools, e.g. story diagrams, to describe the graph transformation systems, the PCM, execution environments like SCA, and cloud services need to be combined into a complete model-driven framework to validate the approach.