Integrating Run-Time Observations and Design Component Models for Cloud System Analysis*

Robert Heinrich[1], Eric Schmieders[2], Reiner Jung[3], Kiana Rostami[1], Andreas Metzger[2], Willhelm Hasselbring[3], Ralf Reussner[1], Klaus Pohl[2]

[1] Karlsruhe Institute of Technology
{heinrich,rostami,reussner}@kit.edu,
[2] paluno, University of Duisburg-Essen
{eric.schmieders,andreas.metzger,klaus.pohl}@paluno.uni-due.de,
[3] Kiel University
{reiner.jung,hasselbring}@email.uni-kiel.de

Abstract. Run-time models have been proven beneficial in the past for predicting upcoming quality flaws in cloud applications. Observation approaches relate measurements to executed code whereas prediction models oriented towards design components are commonly applied to reflect reconfigurations in the cloud. Levels of abstraction differ between code observations and these prediction models. In this position paper, we address the specification of causal relations between observation data and a component-based run-time prediction model. We introduce a meta-model for observation data, based on which we propose a mapping language to (a) bridge divergent levels of abstraction and (b) trigger model updates.

1 Introduction

Cloud applications are subject to continuous change due to modification of the application itself (e.g., emerging requirements) and its execution environment (e.g., platform, user quantity). Since they more and more rely on third-party services, cloud applications increasingly move out of control of the initial developers. Therefore, they must be observed for quality issues [1]. A way to identify upcoming quality flaws is to observe the system and to conduct predictions based on models that reflect the current system state at run-time (i.e. run-time models [2]). In our research, we consider adaptation and evolution as two mutual, interwoven cycles [1]. Central to this perception is a run-time model that is usable for automatized adaptation and understandable for humans during evolution. Run-time models are often close to an implementation level of abstraction to ease the causal relation between the executed code and the model [3]. Existing run-time observation approaches (cf. [5]) provide event-based data sets on source code level, e.g. service entry and exit events. However, it is useful to describe the application structure with design-time components in prediction models [6,7] to (i)

* This work is partially supported by the DFG (German Research Foundation) in the Priority Programme SPP 1593: Design For Future – Managed Software Evolution.

analyze the effects of component-related run-time reconfigurations [8] (e.g., component migration) on quality and (ii) support engineers during system evolution. Code artifacts do not necessarily reflect design components. Hence, the levels of abstraction deviate between observation events and component-oriented models. In this paper, we propose an approach to specify the causal relation between low-level monitoring data and component-based run-time prediction models. We introduce a meta-model for observation data, based on which we propose the *run-time architecture correspondence meta-model* (RAC). The RAC (a) bridges divergent levels of abstraction by providing a language to define mappings and (b) triggers model updates. This enables updating design-time models by observations to form run-time models adequate for reflecting the reconfigurations. The paper is structured as follows. In Sec. 2, we give examples of change scenarios in a cloud context to discuss the state of the art (Sec. 3) and choose a run-time prediction meta-model (Sec. 4). Sec. 5 introduces the meta-modeled observation data. The RAC is described in Sec. 6. The paper concludes in Sec. 7.

2 Dynamic Change at Run-Time

Requirements on the RAC arise from quality-relevant change scenarios of cloud systems gathered in a literature review [8,9,10,11]. We describe how to observe them hereafter. We choose performance and privacy as two examples of qualities, as motivated in [1], while the approach is basically applicable to various quality properties. A privacy law of the European Union (EU) states that sensitive data must not leave the EU. Therefore, we analyze privacy by the geographical location of software components that keep data (e.g., databases). Scenario *S1* to *S3* refer to deployment changes for solving performance issues, due to better load balancing, however simultaneously may cause privacy issues due to changes in the components' geo-locations. *S4* is a provider-intern change of the cloud configuration and *S5* is a change in the system context. Both affect performance.

S1: Migration removes a deployed component instance (cf. [8]) from one execution container and creates a new instance of the same component on another. Observing migration requires information about the instances itself as well as their deployment contexts. In order to verify the privacy constraint, the geographical location of each execution container must be observed.

S2: (De)-replication [8]; Replication is similar to S1, however, the original component instance is not removed. Thus, incoming requests to services can be distributed among the deployed instances. De-replication removes a replica. Observing (de)-replication is analog to S1 but includes requests to instances.

S3: (De)-allocation [8]; Execution containers may become available for deployment (i.e. allocation) while others disappear (i.e. de-allocation). Observing this addresses the identity of containers, e.g. by IP addresses and URLs.

S4: Resizing [9]; Cloud providers may change their platform configuration at run-time, e.g. in-/decrease CPU speed due to energy efficiency. Observing this strongly depends on the cloud service model. Further reading is given in [9].

S5: Changing usage profile [10]; The usage intensity (i.e. workload) of the application and the user behavior may change. The amount of users concurrently

at the system (closed workload [6]), the users' arrival rate (open workload [6]), and the invoked services are contained in observable user sessions [10].

3 State of the Art

Work on the causal relation between the executed code and run-time models can be classified by the model types. Approaches on *parameterized run-time models*, e.g. [12], map single observed service response times to exactly one run-time model parameter. Migration (*S1*) and resizing (*S4*) may be reflected as single parameters. However, information extracted from multiple events (e.g., user behavior (*S5*)) requires to process event sets rather than single events. Complex events are extracted from event sets, e.g. [13], in order to compute QoS properties. However, aggregating observed events to QoS properties is not sufficient to update the application structure (*S2*). Approaches on *behavioral run-time models*, e.g. [14], exploit observed method traces for generating the states and transitions of behavioral models. Although these approaches are useful for reflecting changes in the system usage profiles (*S5*), they lack capturing the system structure (*S1-3*) and properties of the execution context (*S4*). Approaches on *architectural run-time models* establish the causal relation between the executed system and run-time models by automatically creating structural models from monitoring data (cf. the survey in [15]). These approaches generate models that are close to the reflected system and apply graph transformations to create purpose-oriented views on them or try to reconstruct components from the code [8,11]. All the surveyed approaches create run-time models from executed code, however, do not take into account design-time artifacts. Thus, they neglect information that cannot be gathered from the code, such as logical component structures and boundaries, and execution context configurations. This is a drawback especially in a cloud context which is often not completely observable due to the limited visibility of third-party services and platforms [1]. Further, overdetailed models impede understanding and manipulation by humans during system evolution.

4 Run-Time Prediction Meta-Models

We assume that an initial prediction model already exists at design-time by probably making assumption for properties not yet available. This design-time model then turns into a run-time prediction model by updating the model via observation data. Hence, combining design-time and run-time properties is straightforward since they rely on the same meta-model. In our approach, we apply the Palladio Component Model (PCM) [6] as a run-time prediction meta-model. The PCM is tailored to component architectures and provides all the modeling constructs to reflect the aforementioned scenarios, except for geo-location. However, it is straightforward to support geo-location by adding an attribute to execution environment model elements. In contrast, general-purpose prediction formalisms, e.g. LQN and QPN [16], do not provide the specific modeling constructs for component architectures. There are several meta-models related to

the PCM, such as the Descartes Meta-Model (DMM)[7], and those surveyed by Koziolek [17]. They are parameterized to explicitly capture the influences of the components' execution context [7]. We choose the PCM because it is established in the community and offers matured tooling.

5 Measurement Meta-Model

Monitoring frameworks provide system and application level monitoring data in form of single value measurements or more complex data sets [5,18]. These measurements are stored for later aggregation, transformation, and analysis.

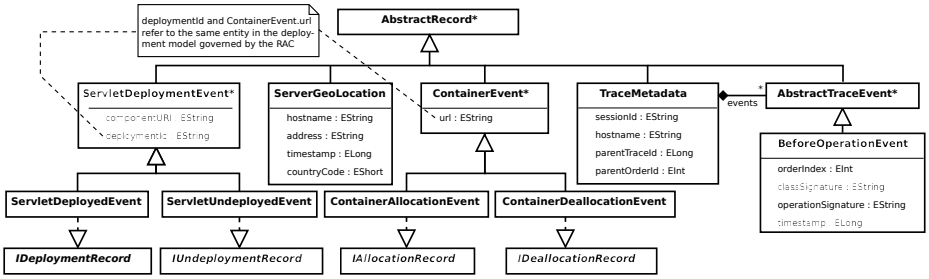

Fig. 1. Excerpt of the MMM Based on the Instrumentation Record Language (IRL) [19]. Interface Names are in *Italics* and Abstract Classes Have an Asterix (*).

The different record types, used by monitoring frameworks, can be seen as a *measurement meta-model* (MMM) where the attributes of the record types are determined by technological limits, the quality properties, and our change scenarios. *S1* and *S2* are based on the deployment and undeployment of components, including, if necessary, the transfer of state. As components, like web-servers and servlets, are technology dependent, they have different kinds of identity information, which require dedicated records. To be still able to distinct deployment and undeployment, we defined two common marker interfaces – IDeploymentRecord and IUndeploymentRecord (cf. Fig. 1). Following the same approach, we provide also marker interfaces for allocation and deallocation for *S3*. Resizing, as defined in *S4*, is presently not covered by its own record type, as it is often not a directly observable event and must be derived from other events. *S5* depends on the observation of operations (i.e. service calls) which is covered by a set of record types based on AbstractTraceEvent and a record holding common trace information (TraceMetadata). For *S5* these records are filtered for entry level calls to construct entry level call sequences.

As many measurements depend on technology, we require a monitoring framework and measurement meta-model designed for extendability. The framework must be fast, reliable, and with low overhead, to ensure that continuous monitoring does not affect the operation of the software. Furthermore, measurements must be accessible in a fast way to support a timely analysis for the operators. In our work, we selected the Kieker framework [18] as it fulfills these requirements [5] and provides a technology independent record notation [19].

6 Run-Time Architecture Correspondence Meta-Model

The MMM exhibits a flat (i.e. non-hierarchical) structure where all records are contained in a large collection and distinguished only by their type and their attributes. Thus, the RAC (cf. Fig. 2) defines a language to describe the causal relation between records of the MMM and elements of a run-time prediction meta-model, here the PCM. A Relation element describes a unidirectional mapping between one or more AbstractRecords of the MMM and a certain PCMElement. The AbstractRecords used in the mapping must fulfill a particular Constraint which is expressed with respect to the AbstractRecord types and their attributes, respectively. Each mapping is further specified by certain formal Rules and a corresponding function to aggregate information, as exemplified hereafter.

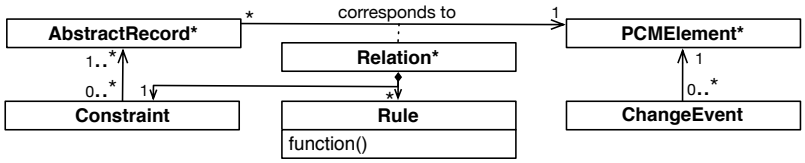

Fig. 2. The Run-Time Architecture Correspondence Meta-Model.

Once an instance of the RAC has been created, a procedure recognizes updates in the associated MMM instance. With any changes to the MMM instance the procedure follows the mappings specified in the RAC instance to update the related PCM instance by throwing a ChangeEvent. A ChangeEvent specifies the PCM element(s) and attribute(s) to be modified and the related updated properties. The ChangeEvent is received by a model update mechanism that is responsible to adequately change the model. Afterwards, the updated PCM instance is applied to predict upcoming quality flaws using existing solvers [6].

Next, we exemplify one Relation of the RAC for open workload (cf. *S5*). The Relation determines the PCM element OpenWorkload from the corresponding BeforeOperationEvents and TraceMetadata, that fulfill the Constraint specified as BeforeOperationEvent.orderIndex is 0 and TraceMetadata.parentTraceId is null. The mapping Rule is specified by the following function. We group the observed traces by their unique TraceMetadata.sessionId. Let τ_i be the least BeforeOperationEvent.timestamp for the i-th TraceMetadata.sessionId. The inter-arrival time for the two successive entries is $\tau_{i+1} - \tau_i$, $\forall i \in \mathbb{N}$. Let T be a random variable representing the inter-arrival rate, the probability distribution function of T can be estimated. The ChangeEvent contains the OpenWorkload element and its interArrivalTime property.

7 Conclusion

We addressed the observation of change scenarios in a dynamic cloud context and proposed the RAC to (a) specify the causal relations between code observation outcomes (MMM) and the corresponding component-based run-time prediction model (PCM), and (b) to trigger run-time prediction model updates.

Future work includes the implementation of the RAC and related tooling, and the development of a model update mechanism that is triggered by change events. Previously, we will complete the records for observing the change scenarios, as specified in the MMM. This includes the investigation of scenarios that are presently not represented by its own record type, such as *S4*. We plan to evaluate the RAC and the tooling by observing changes to a real-life cloud application and updating the corresponding run-time prediction model.

References

1. Hasselbring, W., et al.: iObserve: integrated observation and modeling techniques to support adaptation and evolution. Technical Report 1309, Kiel University (2013)
2. Morin, B., et al.: Models@run.time to support dynamic adaptation. IEEE Computer **42**(10) (2009) 44–51
3. Vogel, T., Giese, H.: Adaptation and abstract runtime models. In: SEAMS, ACM (2010) 39–48
4. Amoui, M., et al.: Software evolution towards model-centric runtime adaptivity. In: CSMR, IEEE (2011) 89–92
5. Eichelberger, H., Schmid, K.: Flexible resource monitoring of java programs. Journal of Software Systems **93** (2014) 163–186
6. Becker, S., et al.: The Palladio component model for model-driven performance prediction. JSS **82** (2009) 3–22
7. Brosig, F., et al.: Modeling parameter and context dependencies in online architecture-level performance models. In: CBSE, ACM (2012) 3–12
8. von Massow, R., et al.: Performance simulation of runtime reconfigurable component-based software architectures. In: ECSA. Volume 6903 of LNCS., Springer (2011) 43–58
9. Frey, S., Hasselbring, W.: The CloudMIG approach: Model-based migration of software systems to cloud-optimized applications. JAS **4**(3 and 4) (2011) 342–353
10. van Hoorn, A., et al.: Generating probabilistic and intensity-varying workload for web-based software systems. In: SIPEW. LNCS, Springer (2008) 124–143
11. Brosig, F., et al.: Automated extraction of arch.-level performance models of distributed component-based systems. In: ASE. (2011) 183–192
12. Canfora, G., et al.: A framework for QoS-aware binding and re-binding of composite web services. JSS **81**(10) (2008)
13. Michlmayr, A., et al.: End-to-end support for QoS-aware service selection, binding, and mediation in VRESCo. IEEE TSC **3**(3) (2010) 193–205
14. van der Aalst, W., et al.: Time prediction based on process mining. Information Systems **36**(2) (2011) 450–475
15. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. SoSyM (December 2013)
16. Kounev, S.: Performance modeling and evaluation of distributed component-based systems using queueing petri nets. IEEE TSE **32**(7) (2006) 486–502
17. Koziolek, H.: Performance evaluation of component-based software systems: A survey. Perform. Eval. **67**(8) (2010) 634–658
18. Hoorn, A., et al.: Kieker: A framework for application performance monitoring and dynamic software analysis. In: ICPE 2012, ACM (2012) 247–248
19. Jung, R., et al.: Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications. In: KPDAYS. (2013) 99–108