An Architectural Model-Based Approach to Quality-aware DevOps in Cloud Applications*

Robert Heinrich[1], Reiner Jung[2], Christian Zirkelbach[2],
Wilhelm Hasselbring[2], Ralf Reussner[1]

[1] Karlsruhe Institute of Technology, Germany,
{heinrich, reussner}@kit.edu
[2] Kiel University, Germany,
{reiner.jung, zirkelbach, hasselbring}@email.uni-kiel.de

Abstract: Cloud-based software applications are designed to change often and rapidly during operations to provide constant quality of service. As a result the boundary between development and operations is becoming increasingly blurred. DevOps provides a set of practices for the integrated consideration of developing and operating software. Software architecture is a central artifact in DevOps practices. Existing architectural models used in the development phase differ from those used in the operation phase in terms of purpose, abstraction, and content. In this chapter, we present the iObserve approach to address these differences and allow for phase-spanning usage of architectural models.

1 Introduction

Cloud Computing technologies have been developed for storing and processing data using distributed resources which are often located in third party data centers. Constructing software systems by incorporating and composing cloud services offers many advantages like flexibility and scalability. Still, considerable challenges come along with these technologies such as increased complexity, fragility and changes during operations that are unforeseeable at development time. As cloud-based systems are designed to change rapidly, they require increased communication and collaboration between software developers and operators, a strong integration of building, evolving and adaptation activities, as well as architectures satisfying deployability in heterogeneous contexts.

DevOps is an umbrella term of practices for enabling software developers and operators to work more closely and thus reducing the time between changing a system and

* This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution and the MWK (Ministry of Science, Research and the Arts Baden-Württemberg in the funding line Research Seed Capital (RiSC).

Preprint of: Robert Heinrich, Reiner Jung, Christian Zirkelbach,Wilhelm Hasselbring, Ralf Reussner (2017):
"An Architectural Model-Based Approach to Quality-Aware DevOps in Cloud Applications".
In: Software Architecture for Big Data and the Cloud. Elsevier, Cambridge, pp. 69-89. ISBN 9780128093382.

putting the change into normal production, while ensuring high quality (Bass et al. 2015). DevOps practices contribute to an integration of the roles of developer and operator. Software architecture is an essential artifact for both, developers and operators. The phase-spanning consideration of software architecture is foundation of DevOps practices. Besides life cycle processes and responsibilities, DevOps practices have strong impact on the software architecture. New architectural styles such as microservices (Newman 2015) emerged to satisfy needs for scalability, deployability and continuous delivery.

By merely introducing new architectural styles, however, the actual problems in collaboration and communication among developers and operators are not solved. Existing architectural models used in the development phase differ from those used in the operation phase in terms of purpose (finding appropriate design vs. reflecting current system configurations), abstraction (component-based vs. close to implementation level), and content (static vs. dynamic). Consequences of these differences are limited reuse of development models during operations and limited phase-spanning consideration of the software architecture.

In this chapter, we propose the iObserve approach for the reuse of architectural development models of cloud-based software applications during the operation phase. We enrich and update the development models with operational observations to construct architectural runtime models. A technology-independent monitoring approach is applied for operational observation. iObserve maintains the semantic relationships between monitoring outcomes and architectural models. We introduce a cloud-based software application as an illustrative example in Sec. 2 before we describe current differences in architectural models among development and operations in Sec. 3. An overview of the iObserve approach is given in Sec. 4. We describe concepts of the iObserve approach to address the differences in architectural models in Sec. 5. A megamodel integrates development models, code generation, monitoring, runtime model updates, as well as adaptation candidate generation and execution. The combination of descriptive and prescriptive architectural models improves the communication and collaboration between operators and developers once a software system is in operation phase. The consideration of static and dynamic content in architectural models supports operation-level adaptations. The application of these concepts is described in Sec. 6. We mention limitations of iObserve in Sec. 7. Related work is discussed in Sec. 8. The chapter concludes with a summary and listing of future work in Sec. 9.

2 A Cloud-based Software Application

We use an illustrative example in this chapter built upon an established community case study – the Common Component Modeling Example (CoCoME) (Herold et al. 2008) – and an associated evolution scenario (Heinrich et al. 2015a). CoCoME resembles a trading system as it may be applied in a supermarket chain. It implements processes at a single cash desk for processing sales, like scanning products or paying, as well as enterprise-wide administrative tasks, like ordering products or inventory management. The detailed design and implementation of CoCoME is described in (Heinrich et al.

2016) and the source code is available for download[1]. We refer to the Java Enterprise implementation of CoCoME in this chapter.

CoCoME uses a database service hosted on data centers that are distributed around the globe, as shown in Fig. 1. The figure illustrates the CoCoME core application and the global reach of prospective cloud providers that offer Database-as-a-Service (DBaaS). During development, architectural models are created and analyzed for quality aspects like performance, e.g. using the Palladio approach for software architecture modelling and simulation (Reussner et al. 2016). If an appropriate design has been found, the system is implemented into source code and deployed on the cloud. This is the point where developers hand over the system to operators to put it in production.

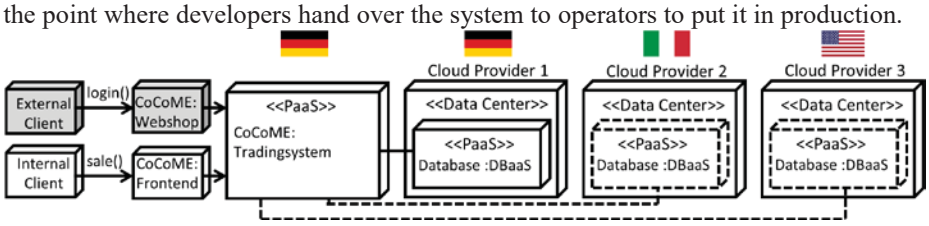

Fig. 1: Actual (solid line) and conceivable (dashed line) component deployment of a cloud-based software application within a global reach of prospective data centers

During system operations, an advertisement campaign of the supermarket chain leads to an increased amount of sales and thus to variations in the application's usage profile and intensity. Increased usage intensity causes an upcoming performance bottleneck due to limited capacities in the given service offering of the cloud provider currently hosting the database. For the sake of simplicity we assume each cloud provider owns exactly one data center. Migrating or replicating the database from one data center to another may solve the scalability issues. Conceivable component deployments are illustrated by dashed lines in Fig. 1. However, migrating or replicating the database may cause privacy issues if sensitive data are transferred outside the European Union (Heinrich et al. 2015a). Privacy has been analyzed for CoCoME in (Schmieders et al. 2014, Schmieders et al. 2015). In cloud-based software applications there is often a trade-off between performance and privacy as further discussed in Sec. 5.2. These privacy issues cannot be foreseen at development time as prospective cloud providers are unknown. In order to analyze upcoming quality flaws during operations and react on them, operators need to observe the system and run analyses based on a model that represents the current application configuration and usage during operations. This model is called an architectural runtime model (Heinrich 2016). However, there are often differences between architectural models used in development and operations which hamper a phase-spanning consideration of the architecture as discussed based on the example in the following.

[1] https://github.com/cocome-community-case-study

3 Differences in Architectural Models Among Development and Operations

This section discusses three differences in architectural models among development and operations – the level of abstraction, the use of prescriptive and descriptive models, and the differences in static and dynamic content reflected by the architectural models.

There are **different abstraction levels** of architectural models in development and operations. Architectural models especially in early phases of development commonly adhere to a component-based paradigm (e.g., Szyperski 2002, Hasselbring 2002). Architectural models used in operations are closer to an implementation level of abstraction. In the CoCoME example, developers specify software components and their interactions in architectural models. Component-based models allow for keeping track of the architecture and structure the system by encapsulated components interconnected via interfaces. During operations the system is observed and architectural models are created from monitoring data. This monitoring data is related to source code artifacts (e.g., service calls or class signatures). For example, observing the sales service of CoCoME (cf. `sale()` in Fig. 1) results in monitoring records for the service itself and all invoked internal services. In addition, the class signature is recorded per service. Yet, no information about the component structure is provided by monitoring. Models derived from the data are close to implementation level, e.g. depict the dependencies between invoked services. Thus, it is hard to reproduce development component models from monitoring data as knowledge about the initial component structure and component boundaries is missing. This knowledge is important for system comprehension and reverse engineering.

For supporting DevOps practices it is useful to **combine prescriptive and descriptive architectural models**. However, a combination of both kinds of models can seldom be found. Prescriptive architectural models are employed during development to document the system to be designed and implemented. During operations, descriptive architectural models are used to reflect the actual state of the running system. Thus, descriptive architectural models are again often created from observation data. Currently there is no phase-spanning notion of software architecture which impedes the combination of prescriptive and descriptive models. In the CoCoME scenario a prescriptive model may be applied during development to make early quality predictions or to make quality predictions for evolutionary changes conducted by developers. An example of an evolutionary change is adding a web shop component (highlighted grey in Fig. 1) to the trading system (Heinrich et al. 2015a). Such a change will have strong impact on various quality properties like privacy, security, performance, and reliability. Developers may want to analyze such quality properties in advance before implementing the change. Descriptive architectural models are applied in the CoCoME example during operations to describe the current system state, potentially after adaptations to the system, e.g. the number of replicated components or the actual geographical location of a migrated component. Developers can modify the descriptive models according to evolutionary changes to construct prescriptive models for quality analysis.

There is **different content (static vs. dynamic)** in architectural models used in development and operations. Architectural models are applied during development to describe the static software design and structure. During operations, architectural models show dynamic content like object stacks in memory, service utilization, and response times of services. Visualization of dynamic content allows the operators to investigate current bottlenecks, analyze for anomalies, and support the decision process for human intervention. In the CoCoME example, a development view of the architecture may comprise the component types, package structure, and class declarations of the application. The operations view may contain charts of resource consumption for the several parts of the architecture and events occurred during operations visualized in sort of a dashboard.

4. The iObserve Approach

The iObserve approach (Hasselbring et al. 2013, Heinrich 2016) specifies operation-level adaptation and development-level evolution as two mutual, interwoven processes that affect each other. Fig. 2 gives an overview of iObserve. The figure is inspired by Oreizy et al. 2008. The evolution activities are performed by human developers, while the adaption activities are executed automatically by predefined procedures where possible without human intervention.

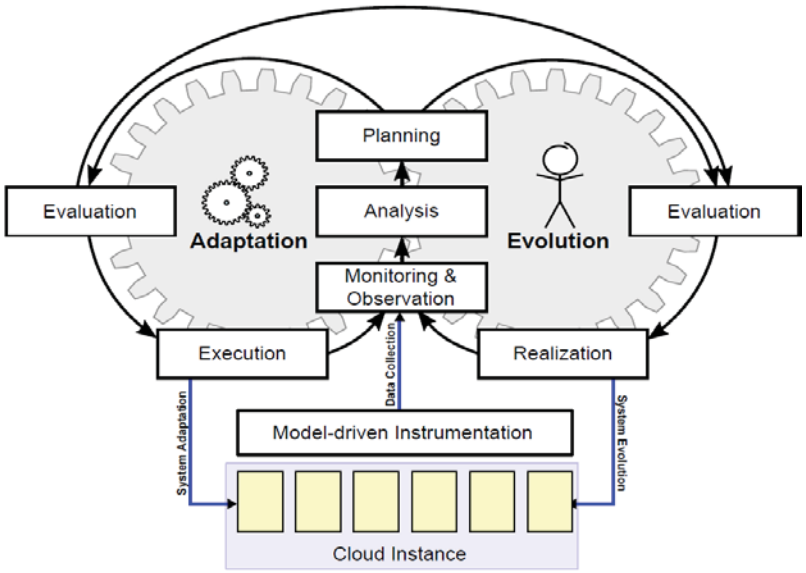

Fig. 2: Overview of the iObserve approach

iObserve tackles architectural challenges in DevOps by following the MAPE (Monitor, Analyze, Plan, Execute) control loop model. MAPE is a feedback cycle commonly used for managing system adaptation (Brun et al. 2009). The MAPE loop is extended with models and transformations between them to facilitate the transition between op-

eration-level adaptation and development-level evolution. The executed software application is observed and used to update the architectural runtime model. Based on the up-to-date model, the current application configuration is analyzed to reveal anomalies and predict quality flaws. The architectural runtime model is then applied as input either for adaptation or evolution activities depending on the outcome of a planning step. In the adaptation process an adaptation plan is selected and evaluated to mitigate deviations. Finally, the plan is executed to update the application architecture and configuration. In the evolution process changes are designed, evaluated and implemented by human developers.

iObserve applies an architectural runtime model that is usable for automatized adaptation and is simultaneously comprehensible for humans during evolution (Heinrich et al. 2014). Foundation is a model-driven engineering approach (Hasselbring et al. 2013) that models the software architecture and deployment in a component-oriented fashion and generates the artifacts to be executed during operations. Therefore, iObserve relies on the Palladio Component Model (PCM) (Reussner et al. 2016) as an architecture meta-model. The PCM consists of several partial meta-models reflecting different architectural views on a software application. The repository model describes components and their interfaces stored in a repository. The components' inner behavior is described in so-called service effect specifications. The system model specifies the software architecture by composing components from the repository. The resource environment model provides a specification of the processing resources (CPU, hard disk and network) while the allocation model specifies the deployment of the components to the resources. The usage model describes the user behavior and usage intensity. Quality-relevant properties, like resource demands of actions and processing rates of resources, are part of the models. Changes during operations relevant in the iObserve context affect the application usage and deployment. In particular, we focus on changes in user behavior and usage intensity, migration and (de)-replication of components, and (de)-allocation of execution contexts (Heinrich 2016). These changes are reflected in the PCM by modifying the usage model and allocation model.

The PCM in its current form is focused on single quality aspects, like performance and reliability, yet does not reflect, for instance, privacy aspects which are relevant in the scope of iObserve. For enabling a more comprehensive representation of quality aspects in the PCM we apply a meta-model modularization and extension approach (Strittmatter et al. 2015) for component-based architecture description languages. Following a reference architecture the information to be represented in the architecture meta-model is divided into four dimensions – paradigm, domain, quality and analysis – as depicted in Fig. 3. Each rectangle with a register symbol in the figure represents a modular meta-model that extends another meta-model and can itself be extended. Each rectangle without a register symbol represents a class or attribute that extends a meta-model. The paradigm layer (π) defines a foundational structure without any semantics, e.g. object oriented design or componentization. Here components and interfaces are specified as core entities. Further, composition by connectors is specified. The domain layer (Δ) extends π and assigns domain-specific semantics to its abstract first class entities. In the context of the iObserve approach Δ will capture software systems whereas

in general any Δ layer is possible, e.g. for embedded or mechatronic systems. For software systems Δ introduces the modules software components and environment. The control flow module extends software components by abstraction of the component behavior similar to flowcharts. Additionally, the software components module is enriched with information whether a component is source or sink of a dataflow. Dataflows are represented by inter-component communications via service calls. The quality layer (Ω) defines the inherent quality aspects of Δ concepts. It contains primarily second class entities, which enrich the first class entities of Δ. In the iObserve approach, the Ω layer comprises performance (light grey) and privacy (dark grey) aspects visualized in Fig. 3. For performance modeling the control flow module is extended with performance-relevant annotations for resource demands of actions and processing rates of resource containers within the execution environment. The performance metrics module contains the meta-modelled metrics and corresponding units. For privacy modeling the dataflows together with information about the geographical location of resource containers form the basis for privacy analysis. The privacy metrics module comprises meta-modelled privacy policies. The analysis layer (Σ) is required if models are used for analyses or simulation. In the iObserve context the performance results module contains a meta-model of service response times. The performance configuration module covers the model-based representation of analysis settings like simulated time span and number of measurements. Analogously, the privacy results module comprises a meta-model of privacy checks including their results. The privacy configuration model covers settings for a privacy check like sensitivity information for data and a blacklist or whitelist of geographical locations. Technical details of privacy analyses conducted in the iObserve context are given in (Schmieders et al. 2014, Schmieders et al. 2015).

Applying the reference architecture enables us adding all iObserve-specific content in a modular and non-invasive way without bloating the PCM. Further details on the application of the reference architecture to the PCM are given in (Strittmatter et al. 2015).

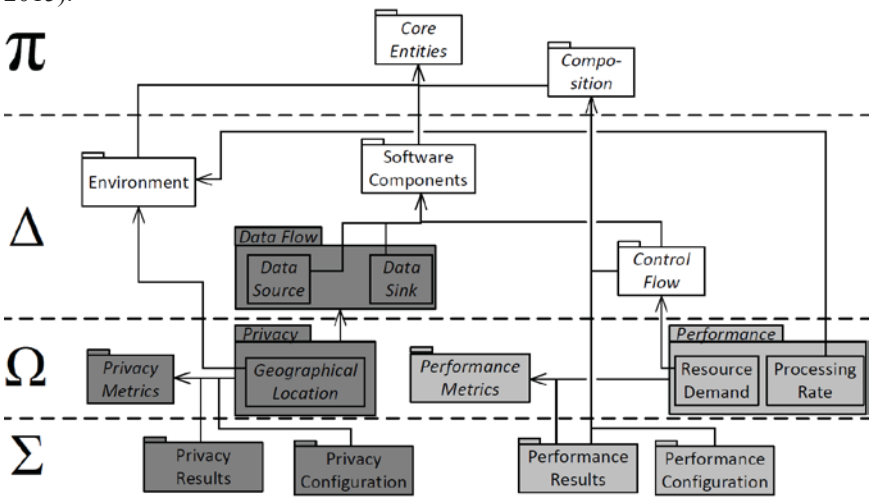

Fig. 3: Exemplary instantiation of the reference architecture for performance and privacy

The cloud application is instrumented with monitoring probes to keep it causally connected with the architectural runtime model. As technical basis for instrumentation, we choose the fast and reliable Kieker monitoring framework (v. Hoorn et al. 2012).

5. Addressing the Differences in Architectural Models

In this section, we describe concepts provided by the iObserve approach to address the three kinds of differences in architectural models among development and operations. The application of the concepts is demonstrated using the CoCoME example in Sec. 6.

5.1 The iObserve Megamodel

The iObserve approach applies a megamodel to bridge the divergent levels of abstraction in architectural models used in development and operations. Megamodels describe the relationships of models, meta-models and transformations (Favre 2004). The iObserve megamodel depicted in Fig. 4 serves as an umbrella to integrate development models, code generation, monitoring, runtime model updates, as well as adaptation candidate generation and execution. Fig. 4 extends a previously published megamodel (Heinrich 2016) by models and transformations for planning and execution. Rectangles depict models and meta-models respectively. Solid lines represent transformations between models while diamonds indicate multiple input or output models of a transformation. Dashed lines reflect the conformance of a model to a meta-model, and, in case of implementation artifacts, the instance of relationship between operations data and development data types.

The iObserve megamodel exhibits four sections defined by two dimensions: one for development vs. operations, and one for model vs. implementation level. On the development side at model level, the megamodel depicts the combination of an architectural model with our model-driven monitoring approach (Jung et al. 2013). The monitoring approach comprises an instrumentation record language (IRL) to define the data structures used for monitoring in a record type model. Further, the monitoring approach comprises an instrumentation aspect language (IAL) to specify the collection of data and the probe placement in an instrumentation model. The architectural model, the record type model, and the instrumentation model are applied for generating source code artifacts of the application and the corresponding monitoring probes. On the operations side at model level, monitoring data that adheres to source code artifacts like Java classes is associated with the elements of the architectural runtime model. Thus, the iObserve megamodel enables the reuse of development models during operation phase by updating them based on operational observations. Moreover, the operation side shows the generation of adaptation candidate models and the adaptation plan construction.

At implementation level, the megamodel depicts development and operations artifacts. The development artifacts comprise the generated implementation of the record types, the instrumentation aspects which implement the probes, and the technology specific artifacts which implement the application, like Servlets and Enterprise Java Beans

in the Java Enterprise context. The operations side shows the monitoring data and their aggregation together with an execution plan describing the precise steps for adaptation on implementation level.

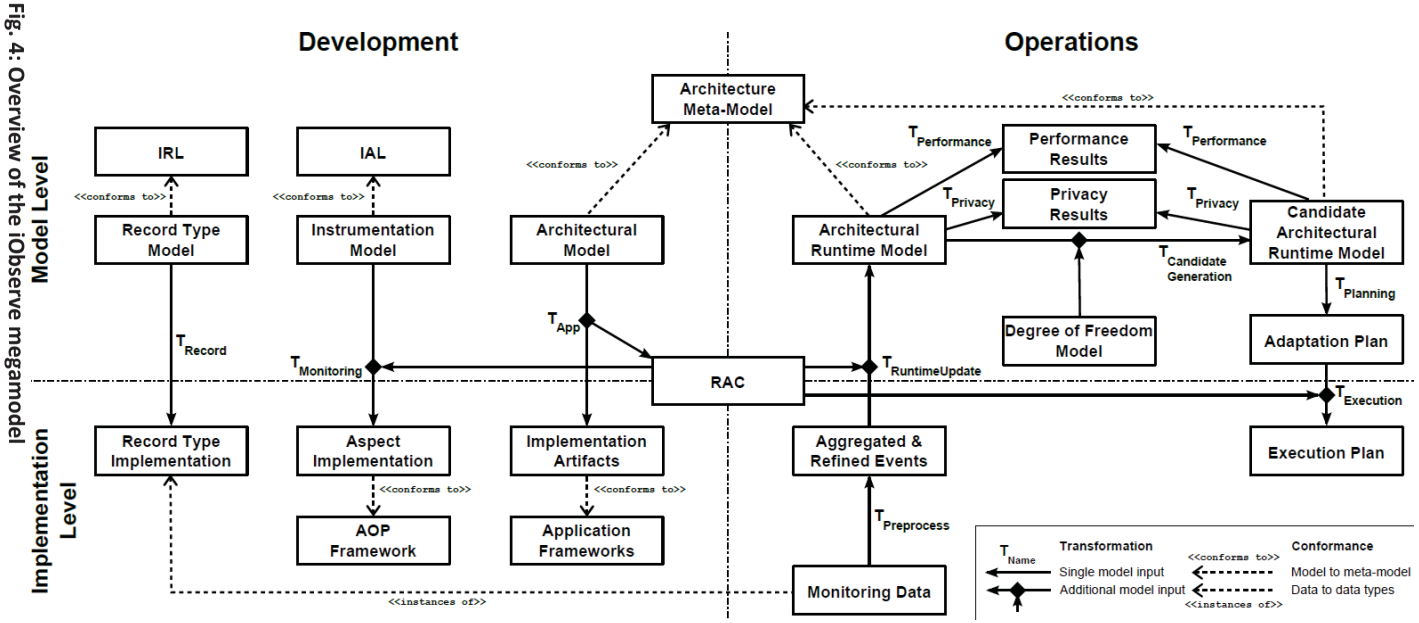

Fig. 4: Overview of the iObserve megamodel

The Runtime Architecture Correspondence Model (RAC) is the central element of the megamodel and crucial for the use of an architectural model at development and operations. The RAC relates architectural model elements to implementation artifacts, like classes and services. It is created during code generation by the transformation T_{App}, as depicted in Fig. 4, or may be specified by hand in scenarios where the code is implemented by a developer. The RAC is also used for the generation and configuration of probes by our model-driven monitoring approach. Monitoring data is a continuous data stream which may comprise large amounts of events, i.e. millions of events per second in large enterprise applications (Fittkau 2012). iObserve first filters and aggregates the monitoring data ($T_{Preprocess}$). Then, iObserve allocates the monitoring data to architectural model elements, and finally uses the aggregated information to update the architectural runtime model by the transformation $T_{RuntimeUpdate}$.

Therefore, the architectural runtime model relates development and operation phases. It allows for phase-spanning consideration of software architecture. Furthermore, it enables quality analyses based on the architecture specification and thus contributes to quality-aware DevOps. Since we update development models by operational observations, our models contain all design decisions, e.g. about component boundaries and the distribution of the application to several execution containers.

Further, in iObserve, the level of abstraction of the initial architectural model and the updated model is maintained, due to:

a) both, the initial architectural model and the architectural runtime model, rely on the same meta-model (the PCM),
b) the decomposition of a development model element in one or more source code artifacts is recorded in the RAC during code generation, and
c) restored while transforming monitoring events related to the source code artifacts to the component-based architectural runtime model.

In consequence, using the iObserve approach changes in the operations phase can be seamlessly integrated with evolutionary changes in the development phase. Operators get access to higher level abstractions through which to view and manipulate the running application while developers can integrate tightly with adaptations that have been made for operational reasons.

5.2. Descriptive and Prescriptive Architectural Models in iObserve

The iObserve approach applies descriptive and prescriptive architectural runtime models for realizing the MAPE control loop as depicted in Fig. 5. In the Monitor phase, iObserve uses information gathered by probes to maintain the semantic relationship between the descriptive architectural runtime model and the underlying cloud application. Descriptive architectural runtime models are applied in the Analyze phase to reveal quality flaws like performance bottlenecks or violations of privacy policies and thus trigger adaptations. If a performance or privacy issue has been recognized, adaptation candidates are generated by the transformation $T_{CandidateGeneration}$ in form of candidate architectural runtime models in the Plan phase as depicted in Fig. 4. These prescriptive candidate models are generated based on a degree of freedom model that specifies variation points in the software architecture. We apply the PCM-based design

space exploration approach PerOpteryx (Koziolek et al. 2011) to the architectural runtime models to find adaptation candidates and rank them regarding quality aspects like performance and costs. PerOpteryx provides a Pareto frontier of optimal design candidates.

Trade-offs between various quality aspects must be considered while planning for adaptation. In the cloud context there is often a trade-off between performance, costs, and privacy. The application usage effects on the performance of the application. Elasticity rules trigger the migration or replication of software components among geographically distributed data centers. Both, migration and replication, may increase performance, however, may lead to violation of privacy policies and increasing costs. We apply PerOpteryx for analyzing trade-offs between performance and costs. Including privacy in design optimization and trade-off analysis is subject of current work (Seifermann 2016).

Once an adaptation candidate has been selected, the model is operationalized by deriving concrete tasks of a plan for adaptation execution. These tasks are derived by the transformation $T_{Planning}$ while comparing a candidate model to the original model and applying the KAMP approach to architecture-based change impact analysis (Rostami et al. 2015). KAMP provides for each change to the architecture elements a set of tasks to implement the change and has already been applied for deriving work plans for solving performance and scalability problems (Heger and Heinrich 2014). The aggregation of the tasks forms the adaptation plan which is transferred in the Execution phase to an execution plan at implementation level by $T_{Execution}$.

In case that no specific model among the candidates can be selected fully automatically, e.g. when there are trade-offs between quality aspects, or if an adaptation plan cannot be derived fully automatically, the human operator (cf. Fig. 5) chooses among the presented adaptation alternatives. Also when no candidate model can be generated, e.g., due to lack of information or criticality of decision, the operator will be involved.

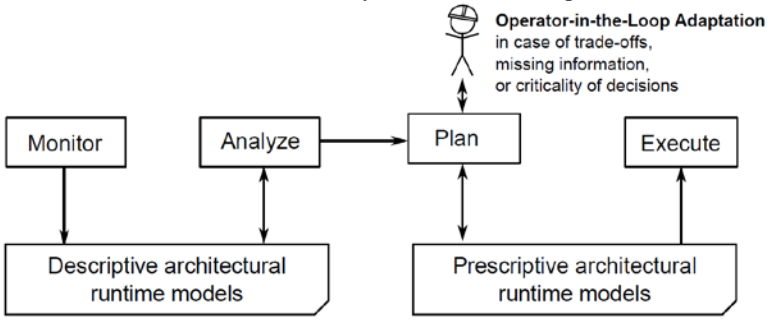

Fig. 5: Descriptive and prescriptive architectural runtime models in the MAPE loop of our operator-in-the-loop adaptation approach (Heinrich et al. 2015b)

5.3. Static and Dynamic Content in Architectural Models

iObserve applies the PCM as an architecture meta-model for modeling usage profiles, the software architecture and deployment as well as quality properties. The PCM is well suited to reflect static system design and structure and due to the extensions of iObserve

it is also able to reflect adaptations during operation phase gathered by monitoring the software system, e.g. component migration or replication. To further improve the support of human operators and thus facilitate operator-in-the-loop adaptation (Heinrich et al. 2015b), iObserve is extended by live visualization of software architectures. Key for live visualization and model inspection (e.g. for performance and privacy) are dynamic attributes of the application, which are collected by our monitoring framework.

Existing live visualization approaches, as listed in (Fittkau 2016), provide interactive visualizations of the deployed application and its internal structure. They monitor and aggregate events during operations to create and update their architectural models. However, they lack static design information, such as component structure, do not provide analyses for privacy, and cannot relate monitoring data to the application architecture defined during development. Consequently, model visualizations of these approaches relate to the implementation level of abstraction. This hinders communication between operators and developers about issues occurred during operations as they use different models.

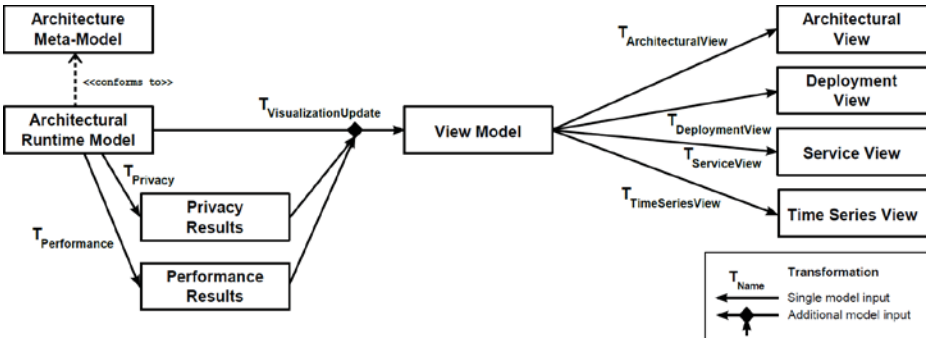

Fig. 6: Extension of the iObserve megamodel for live visualization

In iObserve, we eliminate this drawback by combining static and dynamic content, like in-memory object creation, communication, and execution traces, in an architectural runtime model. This is possible through the mapping capabilities of the RAC which provides model traces relating source code artifacts to architectural model elements. iObserve is inspired by visualization concepts of the live visualization approach ExplorViz (Fittkau 2016). ExplorViz provides two views, called landscape and application level perspective. The first provides an aggregated view on the deployment of software applications solely based on dynamic information. The second allows for viewing the internal structure of a service. The visualizations of iObserve provide similar views. In contrast to ExplorViz, iObserve also allows for utilizing static information which provides further insights into the state of the software application. To provide the information to the operator, iObserve uses a view model depicted in Fig. 6. The view model is created or updated by the transformation $T_{VisualizationUpdate}$ and subsequently used to generate the specific views. In detail, $T_{VisualizationUpdate}$ is triggered after the architectural runtime model has been updated and the results of the performance

and privacy analysis are available. $T_{VisualizationUpdate}$ collects information from the architectural runtime model regarding execution contexts, services, and communication on type and instance level as simple named entities. It associates results of the privacy and performance analyses as well as time series information to these entities, together with other properties, such as call traces, user interaction, and system utilization. While this might seem like a duplication of information, the view model is designed to be a concise representation to support the presentation of different views for the operator. In contrast to the architectural runtime model, the entities do not exhibit any semantics. Instead, they contain information prepared for visualization.

Based on the view model, three views for architecture, deployment, and service are generated by the transformations $T_{ArchitecturalView}$, $T_{DeploymentView}$, and $T_{ServiceView.}$ The views are continuously updated triggered by changes in the view model. Furthermore, the transformations are parametrized by operator selections and are re-executed based on changes to operator selections. Examples based on the CoCoME scenario are given in Sec. 6.3.

To reduce the visual complexity the architectural view and deployment view display multiple instances of a service in an aggregated form. By operator choice, the aggregated services can be selectively expanded showing every service instance. This allows for investigating the state of the software application in more detail. For example, if the response time of a service does not conform to constraints, the aggregated service shows a warning like the box representing the aggregated service is highlighted. The operator can now expand the aggregated view of the service and see all instances. The layout is updated accordingly and only the instances violating the constraint are now highlighted. To further investigate the response time issue, the operator may select the individual service and continue the investigation in the service view. The service view is the adopted application level perspective of ExplorViz, which can now rely on the iObserve view model and present information on a package and component level.

The communication between services is shown as arrows pointing from the caller to the callee. The thickness of the arrow indicates the intensity of the communication, which can be either throughput or number of requests per second. The view model contains both values, while the architecture and deployment view can only visualize one value at a time. In the service view, communication is depicted as lines representing the call traces. The thickness of the lines increase based on number of requests per second.

The three views are limited to the visualization of the current application state including aggregated information on response times and communication intensities. However, in many cases it is important to inspect these values over time. Therefore, execution contexts, services, communication, call traces, and components can be selected and corresponding time series data is displayed in addition to the three views. The time series view is created or updated by $T_{TimeSeriesView}$, which is executed only in case a time series is displayed. The transformation is therefore parameterized based on the selection in one of the three views.

6. Applying iObserve to CoCoME

In this section, we sketch the application of parts of the iObserve approach based on the CoCoME example. We assume code has been generated initially by the transformation T_{App} and the correspondences between architectural model elements and the Java classes have been stored in the RAC. Further, the application is deployed and running.

6.1 Applying the iObserve Megamodel

The observation of the running application using the Kieker monitoring framework produces a stream of heterogeneous events. Following the CoCoME scenario, increased usage intensity of the application triggers changes in the workload specification of the architectural model. iObserve filters out single entry and exit events to services of the application (e.g., the sales service) and aggregates them to sequences of events. The sequences are input to the calculation of the new usage intensity which is then transformed to the PCM workload specification.

The $T_{Preprocess}$ transformation pipeline depicted in Fig. 4 listens to the stream of monitoring events related to entry level services and creates an entry call event for each invocation of the sales service. The transformation aggregates the sales service together with other entry calls (e.g., for reporting or browsing the product catalogue) by exploiting user session information contained in the monitored events. All observed user sessions are combined in a graph-based entry call sequence model to calculate usage-related properties such as path probabilities, loop iterations or usage intensities. The sequence model is input to the $T_{RuntimeUpdate}$ transformation.

$T_{RuntimeUpdate}$ comprises transformations to modify the architectural runtime model according to changes in usage, deployment, and allocation. In the CoCoME scenario, we monitored increased invocations to the sales service which triggers $T_{RuntimeUpdate}$ to modify the workload specification within the PCM usage model. As there might be various usage profiles for different user roles in the model, the transformation takes a look into the RAC to identify the workload specification to be updated for the observed sales service. Note, only the PCM usage model is modified. The other partial models of the PCM (cf. Sec. 4) are taken as they are. If the architectural runtime model is created initially, the other partial models are taken from the architectural model on development side. Otherwise, the models are taken from the existing architectural runtime model. The result of this transformation pipeline is an updated descriptive architectural runtime model.

Similar procedures are applied for observing and processing other changes during operation like migration and (de-)replication of software components and (de-)allocation of execution contexts (Heinrich 2016).

In the CoCoME scenario deployment changes (i.e. migration and (de-)replication) are evoked in the planning phase and therefore already contained in a prescriptive architectural runtime model (cf. Sec. 6.2). Nevertheless, iObserve is capable to observe and processes deployment changes. iObserve first filters out deployment events from the stream of monitoring events. In contrast to aforementioned entry and exit events, a

preprocessing of deployment events is not necessary. A deployment event can be directly mapped to the corresponding resource environment in the architectural runtime model as it contains information about deployed classes and the deployment target (i.e. the resource container). For each deployment event $T_{RuntimeUpdate}$ modifies the resource environment. The RAC is required to identify components corresponding to the observed classes. The same procedure is executed for undeployment events. Moreover, new execution contexts become available (allocation) or existing ones disappear (deallocation) without creating distinct (de-)allocation events. Yet, as a deployment always requires an existing execution context, we can apply the deployment target information contained in the (un-)deployment events to update the resource environment with respect to (de-)allocation. $T_{RuntimeUpdate}$ checks whether (de-)allocation was observed and, if necessary, updates the resource environment before the deployment is updated.

6.2 Applying Descriptive and Prescriptive Architectural Models

After the descriptive architectural runtime model is updated by aforementioned transformations it can be applied for quality analysis. In the CoCoME scenario, the model is analyzed for performance using simulators of the Palladio approach (Reussner et al. 2016). Based on the PCM, including the usage model updated by transformations, the response time distribution for each service of the CoCoME application is simulated. The simulation is depicted as $T_{Performance}$ in Fig. 4 pointing to the performance results model. The simulation indicates upcoming performance bottlenecks caused by increased usage intensity due to the advertise campaign. More precisely, the average response time of the sales services increases by increased usage intensity. For mitigating the performance issues iObserve automatically generates various prescriptive adaptation candidate models by the transformation $T_{CandidateGeneration}$. For candidate generation the degree of freedom in the CoCoME scenario is deployment. Therefore, various candidate models are generated using evolutionary algorithms (Koziolek et al. 2011) each differ in deployment of the database service to data centers. This includes replication and migration of the database service.

During candidate generation, the candidate models are analyzed for performance ($T_{Performance}$) and additionally for privacy ($T_{Privacy}$). For privacy, dataflows are analyzed by constraint checking techniques to ensure that sensitive data do not exceed the EU borders. As CoCoME contains data of different sensitivity we do not peremptorily exclude data centers outside the EU. Data with low sensitivity can be located outside the EU. If privacy violations are identified, the candidate is discarded. This means it is not further evolved to generate new candidates.

Once an appropriate candidate is found the system is adapted based on the prescriptive model ($T_{Execution}$). If no candidate model can be generated based on the given degrees of freedom (e.g., if there are no alternative data center) or if no appropriate candidate model can be identified (e.g., if all candidates that satisfy privacy show performance issues), the human operator is involved for decision making. This is supported by visualization techniques discussed in the next section.

6.3 Applying Live Visualization

The deployment of the CoCoME cloud application is presented in a 2D live visualization in Fig. 7. The visualization has been created from the view model using the transformation $T_{DeploymentView}$. The system border of CoCoME is depicted by the outer box which comprises the execution contexts named WebNode, DataCenter, Adapter, and LogicNode. Inside these execution contexts the single services of CoCoME are shown connected by arrows. Note, the services depicted are composite services each consisting of hierarchically aggregated services. The execution contexts can be virtual machines and dedicated servers, which can be dynamically allocated and deallocated.

The arrow pointing from WebService to CashDesk indicates intensive communication between the two services. Furthermore, the direction indicates that the WebService is calling the CashDesk service. Based on the intensity of the communication, the operator may decide to replicate the CashDesk service. However, before triggering a replication, the operator may want to know more about the service and its communication. Therefore, the operator selects the execution context LogicNode, the CashDesk service, and the TCP connection between WebService and CashDesk to inspect their properties, like the current resource consumption. Consequently, the transformation $T_{DeploymentView}$ is re-executed parametrized by the operator selections. The properties are presented on the right side of Fig. 7 in a small table. In addition, time series data and statistics, such as throughput and response time, can be viewed as time series graphs in the side pane as well by triggering the transformation $T_{TimeSeriesView}$.

To support the operator, the views indicate bottlenecks and violations of constraints by highlighting the specific elements. In Fig. 7, the Inventory service is highlighted indicating a warning level issue based on an SLA violation.

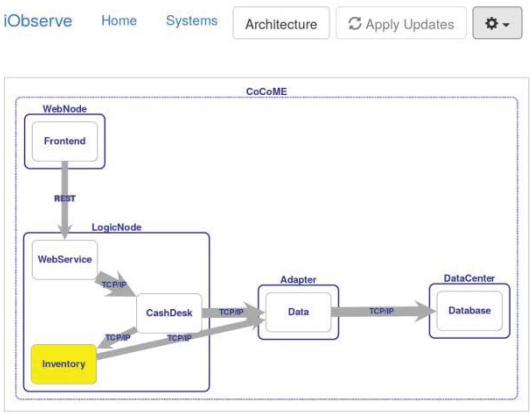

Fig. 7: Deployment view of iObserve depicting a running CoCoME instance

The operator can now consult time series on performance and other properties provided on the side pane, or inspect the interior of the service utilizing the 3D service and component view of ExplorViz depicted in Fig. 8. In the figure, the LogicNode is shown in a 3D live visualization including the Inventory and CashDesk service. The green blocks represent the static package structure of the services, which are alternating colored for better identification. Packages can be opened to show contained classes, like the application.store package, or closed to provide an aggregated view, like the data.store package. The blue pillars visualize dynamic stacks of object instantiations. The communication, based on call traces, is depicted by the orange lines between the classes and packages. Fig. 8 indicates intensive internal communication between classes of the data and application package. This could be the cause of the SLA violation. Based on the visualization, the operator is able to identify potential performance bottlenecks. For further investigation, the operator can inspect the call traces and determine the invoked methods in the classes. If the cause cannot be found on this level, the operator can hand over the information to the developer who then can inspect the related source code, through an integrated source code view or external tools, and the architectural model to identify implementation errors.

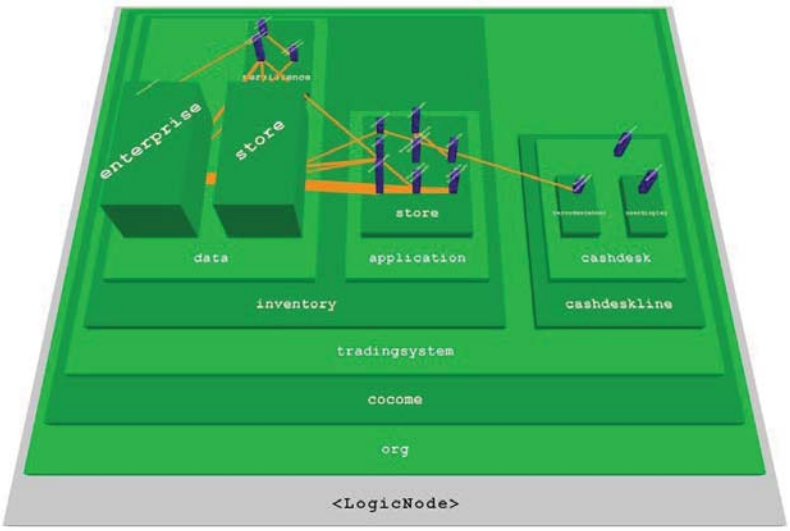

Fig. 8: Service and component view of iObserve visualized in ExplorViz depicting the LogicNode with the Inventory and CashDesk service

7 Limitations

In this section we describe limitations of iObserve identified while applying the approach to the CoCoME community case study.

Currently iObserve is limited to observation and processing of changes in the application's usage and deployment, i.e. migration and (de-)replication of software components and (de-)allocation of execution contexts. These are the most common changes

for cloud-based software applications discussed in literature (Heinrich 2016). iObserve can be extended for additional types of changes easily by adding new monitoring probe specifications and new transformations to the megamodel.

iObserve focuses on the software application architecture and does not consider internal events of the cloud infrastructure. Thus, the impact of infrastructure internals, e.g. changes in the technology stack or internal replications, on the application's quality is not considered. We use a PAAS cloud. Therefore, we assume we can observe all events needed from the perspective of an application developer and operator. Nevertheless, SAAS-based services can be represented in the architectural model and may be supported by additional monitoring technologies in the future development of iObserve.

Increased criticality and limited observability of cloud-based applications require involving humans in the operation and adaptation process (Heinrich et al. 2015b). The iObserve approach supports human engagement at several points in the process. Humans are supported by visualization of the current situation and reveling consequences of design decisions. At the end, the human operator still needs knowledge and experience to make a good decision. However, due to the support given by iObserve decision making is expected to be much easier.

A common limitation of runtime modelling approaches is the accuracy of the model depends on the length of the time span of observation. If the time span is too short, services invoked seldom may not be observed or probabilities calculated may be inaccurate.

8. Related Work

Work related to the concepts proposed in this chapter can be distinguished into four major categories – (i) approaches for reusing development models during operations, (ii) approaches for model extraction from observation data, (iii) approaches for architecture conformance checking, and (iv) approaches for trace visualization.

Work on *reusing development models* during operations (e.g., Morin et al. 2009, Ivanovic et al. 2011, Canfora et al. 2008) employs development models as foundation for reflecting software systems during operations. Bencomo et al. 2014 gives an overview of runtime modeling and analysis approaches. The work in (Morin et al. 2009) reuses sequence diagrams to verify running applications against their specifications. However, the approach does not include any updating mechanisms that changes the model whenever the reflected systems is being alternated. Consequently, changes during operation phase are not supported. Other than this, the runtime models in (Ivanovic et al. 2011) and (Canfora et al. 2008) are modified during operations. These approaches employ workflow specifications created during development phase in order to carry out performance and reliability analyses during operation phase. The approaches update the workflow models with respect to quality properties (e.g., response times) of the services bound to the workflow. However, these approaches do not reflect component-based software architectures. Further, this work updates the model with respect to single parameters and does not change the model's structure.

Work on *model extraction* creates and updates model content during operations. Approaches such as (Song et al. 2011, Schmerl et al. 2006, van der Aalst et al. 2011) establish the semantic relationships between executed applications and runtime models based on monitoring events (for a comprehensive list of approaches see Szvetits and Zdun 2013). Starting with a "blank" model, these approaches create model content during operation phase from scratch, e.g. by observing and interpreting service traces. Therefore, they disregard information that cannot be gathered from monitoring data, such as development perspectives on component structures and component boundaries. For instance, the work in (van der Aalst et al. 2011) exploits process mining techniques for extracting state machine models from event logs. Without knowledge about the component structure created during development, the extracted states cannot be mapped to the application architecture specified in development phase. In consequence, the model hierarchy is flat and unstructured, which hinders software developers and operators in understanding the application at hand. Further, the work reflects processes but neither components nor their relationships. Other than this, the work in (Schmerl et al. 2006) extracts components and their relationships from observations for architecture comparison. With this approach we share the application of transformation rules to update a runtime model based on monitoring events. The resulting model in (Schmerl et al. 2006) is coarse-grained, which is sufficient for their purposes. However, when conducting performance and privacy analyses the observation and reflection of resource consumptions is crucial. Reflecting the consumption by the means of usage profiles requires processing event sets rather than single events, which outruns the capacity of this approach. Further, the observation and analysis of usage and component changes causes complex relationships between the investigated applications, probe types, and runtime models, which is not discussed in (Schmerl et al. 2006).

Work on *architecture conformance checking* compares the static source code of a software application to an architectural model or architectural constraints. An early approach of architecture conformance checking is based on Sotograph and allows for comparing different source code versions with an architectural model to detect architecture degradation (Bischofberger 2004). A unifying approach (Caracciolo et al. 2015) integrates different conformance checking tools and provides a common rule based interface to them. As a rule based approach no explicit architecture model is used. (Passos et al. 2010) investigate three different conformance check approaches based on dependency matrices, source code queries and reflexion models. All these approaches rely only on static information and cannot capture the deployment of a software application. Furthermore, they try to recover the architecture or architectural properties based on the source code without knowledge of the actual relationship of code and architectural model. In iObserve, we use the RAC to ensure that the correct code is related to the architecture and due to operational observations, we are able to include dynamic properties in our analyses.

Related work on *trace visualization* utilizes concepts similar to those proposed in the paper. The visualizations of iOberve are inspired by ExplorViz (Fittkau 2016) and therefore most related to this approach. ExplorViz creates views based on monitoring data, however, neglects development decisions. The TraceCrawler approach (Greevy et al. 2006) visualizes prerecorded program traces relating to a single software service.

Therefore, TraceCrawler implements an offline analysis, while iObserve focuses on live visualization of multiple software systems in a large software landscape. The visualization of TraceCrawler is based on a 3D graph metaphor where each instance of a class is represented by a box and the invocation is represented by edges between these boxes (Greevy et al. 2006). This individual representation of instances and invocations can lead to a complex web of boxes and edges which are hard to comprehend. Therefore, we use an aggregated view, where each class is represented by a single box and edges are drawn between classes instead of instances. The amount of instances is represented through the height of a box, and the number of invocations determines the thickness of edges. The CodeCity approach (Wettel and Lanza 2007) uses a city metaphor for large-scale software systems, like ExplorViz. However, CodeCity is only able to visualize static properties of source code, like classes and packages, where iObserve, ExplorViz, and TraceCrawler include dynamic content. Furthermore, CodeCity visualizes the whole application in class level detail at once. In contrast, we employ a top-down driven hierarchical approach, which allows for inspecting single packages interactively.

To summarize, development models reused during operation phase provide good comprehensibility to humans, but are not updated with respect to structural changes yet. However, structural updates are required to reflect changes during operations like replication or migration as further described in (Heinrich 2016). Work on model extraction automatically creates runtime models from scratch. As development decisions on the application architectures cannot be fully derived from monitoring events the resulting models lack understandability. Approaches on architecture conformance checking neglect dynamic properties as they are limited to the conformance between static source code and architectural models. The visualization of models is essential to support understandability. Most visualization approaches reflect only static models and source code, which cannot provide insight in operational properties. Approaches like TraceCrawler incorporate monitoring data however are not able to provide live visualizations, which are necessary to support operators. ExplorViz offers live visualization but neglects development decisions.

Moreover, there are approaches for model synchronization using triple graph grammars for instance. Trollmann and Albayrak 2016 propose a triple graph for synchronizing Enterprise Java Beans and component models. However, data do not result from monitoring a running application.

9. Conclusion

In this chapter we proposed the iObserve approach to address architectural challenges in DevOps of cloud-based software applications. The iObserve megamodel bridges different abstraction levels among architectural models in development and operations. iObserve employs descriptive and prescriptive architectural runtime models in the context of the MAPE control loop. Extending the iObserve megamodel for live visualizations allows for depicting static as well as dynamic content in architectural models used in operator-in-the-loop adaptation.

In the future, we plan to further investigate the planning and execution phases of iObserve. Besides further analyzing design space exploration and optimization approaches to find optimal architectural runtime model candidates, we will investigate the execution of adaptation plans to allow for a maximum degree of automation where adaptation is possible without human intervention. Our live visualization supports cases where human intervention is required. We will extend the visualization approach by additional 3D views to further improve support of human operators. We will conduct experiments for evaluating our architectural runtime models with respect to fidelity, usefulness for human inspection, and scalability.

Moreover, we will extend and revise the PCM following our reference architecture for meta-model modularization and extension to support additional quality aspects that may become relevant in the iObserve context.

References

W. van der Aalst, M. Schonenberg, and M. Song. Time prediction based on process mining. Information Systems, 36(2):450–475, 2011.

L. Bass, I. Weber, and L. Zhu. DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015

N. Bencomo, R. France, B. H. C. Cheng, and U. Amann. Models@run.time. Springer, 2014.

W. Bischofberger, J. Kühl and S. Löffler. Sotograph – A Pragmatic Approach to Source Code Architecture Conformance Checking, pages 1-9, LNCS, 3047, 2004

Y. Brun et al. Engineering Self-Adaptive Systems Through Feedback Loops, In: Software engineering for self-adaptive systems, pages 48–70. Springer, 2009.

G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. A framework for QoS-aware binding and rebinding of composite web services. Journal of Systems and Software, 81(10):1754–1769, 2008.

A. Caracciola, M. Lungu, and O. Nierstrasz. A Unified Approach to Architecture Conformance Checking. In 12th Working IEEE/IFIP Conference on Software Architecture, pages 41-50, ACM. 2015.

J.-M. Favre. Foundations of model (driven) (reverse) engineering – episode i: Story of the fidus papyrus and the solarus. In Dagstuhl post-proccedings, 2004.

F. Fittkau, S. Frey, and W. Hasselbring. Cloud User-Centric Enhancements of the Simulator CloudSim to Improve Cloud Deployment Option Analysis. In European Conference on Service-Oriented and Cloud Computing, Springer, 2012.

F. Fittkau, A. Krause, W. Hasselbring. Software Landscape and Application Visualization for System Comprehension with ExplorViz, Information and Software Technology. 2016, http://dx.doi.org/10.1016/j.infsof.2016.07.004

O. Greevy, M. Lanza, and C. Wysseier, Visualizing live software systems in 3D. In ACM Symposium on Software Visualization, 2006, pp. 47–56.

W. Hasselbring. Component-Based Software Engineering, In: Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing, Singapore, pp. 289-305, 2002.

W. Hasselbring, R. Heinrich, R. Jung, A. Metzger, K. Pohl, R. Reussner, and E. Schmieders. iObserve: integrated observation and modeling techniques to support adaptation and evolution of software systems. Technical Report 1309, Kiel University, Kiel, Germany, 2013.

C. Heger and R. Heinrich. Deriving work plans for solving performance and scalability problems. In Computer Performance Engineering, volume 8721 of Lecture Notes in Computer Science, pages 104-118. Springer, 2014.

R. Heinrich. Architectural run-time models for performance and privacy analysis in dynamic cloud applications. SIGMETRICS Performance Evaluation Review, 43(4):13-22, ACM, 2016.

R. Heinrich, Kiana Rostami, Ralf Reussner. The CoCoME platform for collaborative empirical research on information system evolution. Technical Report 2016,2; Karlsruhe Reports in Informatics, KIT, 2016.

R. Heinrich, E. Schmieders, R. Jung, K. Rostami, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl. Integrating run-time observations and design component models for cloud system analysis. In 9th Int'l Workshop on Models@run.time, pages 41–46. CEUR Vol-1270, 2014.

R. Heinrich, S. Gärtner, T.-M. Hesse, T. Ruhroth, R. Reussner,K. Schneider, B. Paech, and J. Jürjens. A platform for empirical research on information system evolution. In 27th Int'l Conference on Software Engineering and Knowledge Engineering, pages 415–420. KSI Research Inc., 2015.

R. Heinrich, R. Jung, E. Schmieders, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl. Architectural run-time models for operator-in-the-loop adaptation of cloud applications. In IEEE 9th Symposium on the Maintenance and Evolution of Service-Oriented Systems and Cloud-Based Environments. IEEE, 2015.

S. Herold et al. CoCoME – the common component modeling example. In The Common Component Modeling Example, pages 16–53. Springer, 2008.

A. v. Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In 3rd Int'l Conference on Performance Engineering (ICPE 2012), pages 247-248. ACM, 2012.

D. Ivanovic, M. Carro, and M. Hermenegildo. Constraint-based runtime prediction of SLA violations in service orchestrations. In Service-Oriented Computing, pages 62–76. Springer, 2011.

R. Jung, R. Heinrich, and E. Schmieders. Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications. In Symposium on Software Performance, pages 99-108. CEUR Vol-1083, 2013.

A. Koziolek, H. Koziolek, R. Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. ACM SIGSOFT conference on Quality of software architectures. 2011.

B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. IEEE Computer, 42(10):44–51, 2009.

S. Newman. Building Microservices. O'Reilly. 2015.

P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: Framework, approaches, and styles. In Companion of the 30th Int'l Conference on Software Engineering, pages 899–910. ACM, 2008

L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca. Static Architecture-Conformance Checking: An Illustrative Overview. In IEEE Software, 27(5), 2010.

Reussner, Ralf H. et al., editor. Modeling and Simulating Software Architectures -- The Palladio Approach, MIT Press, 2016. ISBN: 978-0-262-03476-0

K. Rostami, J. Stammel, R. Heinrich, and R. Reussner. Architecturebased assessment and planning of change requests. In 11th International ACM SIGSOFT Conference on Quality of Software Architectures. ACM, 2015, pp. 21–30.

B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. IEEE Transactions on Software Engineering, 32(7):454–466, 2006.

E. Schmieders, A. Metzger, K. Pohl. A Runtime Model Approach for Data Geo-location Checks of Cloud Services, International Conference on Service-Oriented Computing, pp. 306–320, Springer. 2014.

E. Schmieders, A. Metzger, K. Pohl. Runtime Model-Based Privacy Checks of Big Data Cloud Services, International Conference on Service-Oriented Computing, pp. 71-86, Springer. 2015.

S. Seifermann. Architectural data flow analysis. 13th Working IEEE/IFIP Conference on Software Architecture. IEEE. 2016

H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. Journal of Systems and Software, 84(5):711–723, 2011.

M. Strittmatter, K. Rostami, R. Heinrich, and R. Reussner. A modular reference structure for component-based architecture description languages. In 2nd International Workshop on Model-Driven Engineering for Component-Based Systems, pages 36-41. CEUR, 2015.

M. Szvetits and U. Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. SoSyM, 2013.

C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2nd edition, 2002.

Frank Trollmann and Sahin Albayrak. Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models. In Theory and Practice of Model Transformations, LNCS 9765, pp. 91–106. Springer. 2016.

R. Wettel and M. Lanza. Visualizing software systems as cities. In 4th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007), 2007.