Architectural runtime models for integrating runtime observations and component-based models

Robert Heinrich

Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany

A R T I C L E   I N F O

Keywords:
Software architecture
Runtime model
Performance model
Workload
Palladio Component Model

A B S T R A C T

Keeping track of modern software applications while dynamically changing requires strong interaction of evolution activities on development level and adaptation activities on operation level. Knowledge about software architecture is key for both, developers while evolving the system and operators while adapting the system. Existing architectural models used in development differ from those used in operation in terms of purpose, abstraction and content. Consequences are limited reuse of development models during operation, lost architectural knowledge and limited phase-spanning consideration of software architecture.

In this paper, we propose modeling concepts of the iObserve approach to align architectural models used in development and operation. We present a correspondence model to bridge the divergent levels of abstraction between implementation artifacts and component-based architectural models. A transformation pipeline uses the information stored in the correspondence model to update architectural models based on changes during operation. Moreover, we discuss the modeling of complex workload based on observations during operation. In a case study-based evaluation, we examine the accuracy of our models to reflect observations during operation and the scalability of the transformation pipeline. Evaluation results show the accuracy of iObserve. Furthermore, evaluation results indicate iObserve adequately scales for some cases but shows scalability limits for others.

1. Introduction

Modern software systems, such as cloud-based applications, change rapidly to react on the ever increasing amount of events the systems must handle. Examples of events are outage of resources, peak loads during marketing campaigns, emerging requirements and changes of infrastructures or third party services. Changes may comprise the replication or migration of a database component (Heinrich, 2016), replacement of one algorithm implementation by another (Qin and Eichelberger, 2016), or more substantial modifications of the system design like adding a new feature or changing the architecture style (e.g., monolithic system changed to microservices). Keeping track of these changes requires strong interaction of evolution activities on development level and adaptation activities on operation level. Consequently, more collaboration or even integration of the role of developer and operator is proclaimed in emerging paradigms like DevOps. DevOps is a set of practices of developers and operators participating together in the entire application lifecycle, from design through the development process to production support (Mueller, 2019).

E-mail address: robert.heinrich@kit.edu.

Software architecture is a central artifact to keep track of the system for both, developers while evolving the system and operators while adapting the system. Thus, the phase-spanning consideration of software architecture is essential for aligning development-level evolution and operation-level adaptation.

Existing architectural models used in the development phase differ from those used in the operation phase in terms of purpose, abstraction and content (Heinrich et al., 2017). Consequences of these differences are limited reuse of development models during operation, lost architectural knowledge and limited phase-spanning consideration of the software architecture.

In our previous work, we proposed architectural runtime models as a means for combining automated adaptation as well as the human inspection of modern software systems (Heinrich et al., 2015b). While runtime models have shown their effectiveness for self-adaptation, using runtime models during software evolution has been neglected so far (Heinrich et al., 2014). As commonly observed, software systems in operation often drift away from their initial development models (Murphy et al., 2001). In contrast, runtime models are kept in-sync with the underlying system. Thus, runtime models may serve as valuable basis for evolution and adaptation activities (Heinrich et al., 2015b). However, typical runtime models are close to an implementation level of abstraction (Vogel and Giese, 2010). While being useful for self-adaptation, such low level of abstraction impedes system comprehension for human developers when evolving the system. In addition, owing to various modifications during the application's lifecycle, runtime models may grow in detail or become unnecessarily complex, which severely limits comprehensibility of this kind of runtime models for humans during software evolution (e.g., see Vogel and Giese, 2014).

The iObserve approach (Hasselbring et al., 2013) targets to facilitate the automated analysis as well as the human inspection of architectural models by following the MAPE (Monitor, Analyze, Plan, Execute) control loop model for managing system adaptation (Brun et al., 2009). Architectural runtime models are initialized and updated via dynamic processing of monitoring data (Hasselbring, 2011). As an umbrella to integrate development models, code generation, monitoring, analysis and runtime model update, we proposed a concise megamodel in Heinrich (2016). The concept of a megamodel reflects the relationships of models, metamodels and transformations (Favre, 2004; Vignaga et al., 2013). The iObserve megamodel is applied for updating the architectural runtime model based on monitoring data. Subsequently, the updated model is analyzed for quality flaws, like performance bottlenecks in the application (Reussner et al., 2016) and confidentiality issues in data processing (Seifermann et al., 2019; Schmieders et al., 2015), which may trigger adaptation or evolution activities based on the architectural runtime model.

In a previous publication (Heinrich, 2016), we introduced several changes to cloud-based software applications during operation. Contributions of iObserve are on monitoring (Jung et al., 2013) and analyzing cloud-based software applications (Heinrich, 2016). Differences between architectural models among development and operation have been discussed in Heinrich et al. (2017). Based on these differences, the usage of the megamodel in the plan and execution phase of the MAPE loop has been sketched in Heinrich et al. (2017) and implemented in Pöppke (2017).

This paper is a significant extension of our previous work by providing (i) detailed description of the models and transformations of the iObserve megamodel and (ii) evaluation of accuracy and scalability of model construction in the iObserve approach based on observations during operation. After giving an overview of possible changes during operation in Section 2 and introducing the iObserve approach in Section 3, we provide details on our modeling concepts in Section 4. Contributions reported in this paper are as listed hereafter.

- Following an introduction of the iObserve megamodel and the parts relevant in this paper in Section 4.1, we give a detailed description of the Runtime Architecture Correspondence Model (RAC) in Section 4.2 to specify the correspondence between an element of the architectural model and implementation artifacts. Thus, the RAC bridges the divergent levels of abstraction between component-based architectural models and the implementation level. While generating source code from the architectural model, correspondence information is recorded in the RAC and is subsequently used for updating the architectural model based on observations of the executed application.
- In Section 4.3, we extend the transformation pipeline introduced in Heinrich (2016) by transformations to update the architectural runtime model for changes in component deployment and allocation of execution containers like servers.
- The term workload denotes the usage intensity and user behavior an application faces (Reussner et al., 2016). In Section 4.4, we come up with an approach to model complex workload (e.g., several user groups of different behavior and nested user behavior) based on observations during operation. In contrast to related approaches, this approach exploits knowledge from the existing architectural model to identify different user groups, their user behavior and usage intensity. It is fully integrated in the iObserve approach by exploiting the information specified in the RAC to identify user interaction and drive model updates using the transformation pipeline.
- Section 4.5 then discusses the applicability of the proposed modeling concepts in terms of dependencies to tools and effort for conducting human activities in the iObserve approach.
- In Section 5, we evaluate the accuracy of the architectural runtime models for reflecting changes in workload (i.e. user groups, user behavior as well as usage intensity), component deployment and execution container allocation. Moreover, we evaluate the scalability of the transformation pipeline for updating the models to reflect the changes during operation.
- After the features of iObserve have been described and evaluated, Section 6 shows a comparison of the features of iObserve to those of related approaches to point out the novelty of our approach.

The paper concludes with a summary and outlook of future work in Section 7.

2. Changes during operation

Changes during application operation that are relevant in the context of this paper 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 containers (Heinrich, 2016). Hereafter, we briefly describe the changes relevant for understanding the paper and discuss how to identify them while observing a cloud-based application.

Workload characterization changes (C1): The usage intensity of the application and the user behavior may change which may affect the system's quality properties like performance (cf. Khan et al., 2012). The amount of users concurrently using the application (closed workload Reussner et al., 2016), the users' arrival rate at the system (open workload Reussner et al., 2016), and the invoked services are contained in observable user sessions (van Hoorn et al., 2008).

The deployment of the application's software components may change, for example to address performance issues by migration or replication of components. These changes may either be triggered actively by an adaptation mechanism or can be performed autonomically by the observed system. Changes in deployment may cause confidentiality issues in data processing and storage.

Migration (C2): moves a software component from one execution container to another. Migration is typically realized by undeploying the component on one execution container, and creating a new instance of the same component type on another execution container (von Massow et al., 2011). In consequence, migration is in essence the combination of one undeployment and one deployment operation of a component of the same type on different execution containers.

Replication (C3): duplicates a running component instance in a way that the workload can be distributed among the deployed instances. This operation is based on the deployment operation. Replication can only be performed if the architecture allows to distribute requests to a component.

Dereplication (C4): is the inverse operation to C3. It is in essence the undeployment of a component instance which has been replicated before.

In order to distinguish the changes C2 to C4, the events must have unique identifiers. Based on these identifiers, a sequence of deployment and undeployment events of the same component type with the same instance identifier can be clearly identified as migration. In contrast, replication creates a new instance with a new identifier of the same component type. Note, in the following we will not distinguish between component type and instance but merely use the term component for reasons of brevity.

(De)allocation (C5/C6): appears when execution containers become available (allocation), or disappear (deallocation) (von Massow et al., 2011). Although the observation of both changes is highly technology dependent, we described several ways to do so in Heinrich (2016). In the scope of this paper, we focus on systems where deployment and undeployment of components is handled autonomously. For these systems allocation and deallocation is often performed implicitly without creating distinct events. As a deployment always requires an existing execution container, deployment events imply the necessary allocation of an execution container. Analogously, the undeployment event of the last component deployed on an execution container may imply the deallocation of the execution container.

3. Overview of iObserve

This section gives an introduction and conceptual overview of the iObserve approach before modeling details are described in the next section.

Fig. 1 gives an overview of iObserve. The figure is inspired by Oreizy et al. (2008). Parts in the focus of this paper are marked blue. In the iObserve approach, we consider development-level evolution and operation-level adaptation as two mutually interwoven processes (Hasselbring et al., 2013). The evolution activities are conducted manually (i.e. in a non-automated way) by the human developer to implement perfective, adaptive or corrective changes to the system (Lientz and Swanson, 1980). Adaptation activities are performed fully automatically by predefined procedures where possible without human intervention. If human intervention is required, the human operator is involved in adaptation (i.e. operator-in-the-loop adaptation Heinrich et al., 2015b). Central to the iObserve approach is an architectural runtime model that is usable for automatized adaptation and is simultaneously comprehensible for humans in the evolution process.

During the initial development of a software application, the human developer manually creates an architectural model to describe the application's architecture and deployment in a component-based fashion. Based on the architectural model the artifacts to be executed during operation are automatically generated. These artifacts are manually extended by the human developer to create an executable application (see details in the following section).

During the operation of the application, the state of the executed application is observed by monitoring (Jung et al., 2013) and used to automatically update the architectural model. The operator can run automated analyses based on the architectural model to detect upcoming quality flaws (Reussner et al., 2016; Seifermann et al., 2019). The architectural model is used for planning an adaptation to mitigate the quality flaws and finally to execute the adaptation procedure (Heinrich et al., 2017) – where possible without human intervention. For evolution, the architectural model supports the human developer in making design decisions which are evaluated and finally realized by implementing source code and redeploying the application. All the tooling developed in the iObserve project is available for download in the iObserve GitHub repository.[1]

[1] https://github.com/research-iobserve.

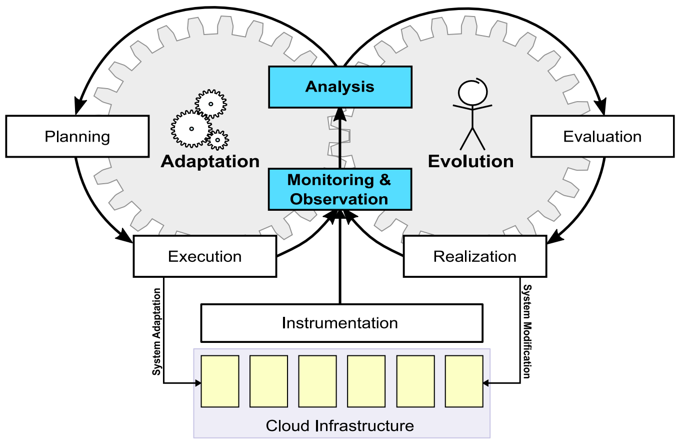


Fig. 1. iObserve cloud application life-cycle: Considering adaptation and evolution as two interwoven processes.

4. Architecture modeling

After describing the iObserve approach on a conceptual level, we take a more technical stance in this section by discussing the models, metamodels and transformations required to realize the approach.

4.1. The iObserve megamodel

In order to keep track of the different models, metamodels and transformations of iObserve, we apply the concept of a megamodel. Megamodels provide a notation for the relationships of models, metamodels and transformations (Favre, 2004; Vignaga et al., 2013). The iObserve megamodel is shown in Fig. 2. The parts of the megamodel in the focus of this paper are marked blue. To give the reader an impression of the complete approach, we sketch the megamodel in full extent in this subsection before we focus on the marked parts in the following subsections. Note, only the most relevant relationships are depicted to keep the visualization of the megamodel understandable.

The iObserve megamodel serves as an umbrella to integrate development models, code generation, monitoring, runtime model update as well as adaptation planning and execution. Rectangles represent models and metamodels, respectively. Solid lines represent transformations between models while diamonds indicate multiple input or output models of a transformation. Dashed lines represent the conformance of a model to a metamodel, of data to a data type or of an artifact to a framework. Previous versions of the iObserve megamodel have been published in Heinrich (2016) and Heinrich et al. (2017).

In contrast to related runtime modeling approaches (discussed in Section 6), iObserve utilizes the same software architecture modeling language at development and operation to provide rich models which are comprehensible for developers and operators likewise, and which can be fed back into the evolution process without the need of conversion and the risk of loss of knowledge. We use the Palladio software architecture modeling language defined by a comprehensive metamodel – the Palladio Component Model (PCM) (Reussner et al., 2016) – as the architecture metamodel (see Fig. 2). The PCM has been chosen because it is established in the software architecture community, offers matured tool support (named the Palladio Bench Reussner et al., 2016), and comprises the modeling constructs to represent the changes described in Section 2. A detailed discussion on the selection of the architecture metamodel is given in Heinrich (2016). The PCM comprises several partial metamodels for describing different architectural views on a software system. The following partial metamodels are relevant for models and transformations described in this paper. A complete overview and further details on the PCM can be found in Reussner et al. (2016). The repository model describes components and their interfaces stored in a repository. The system model specifies the software architecture by assembling components from the repository. The resource environment model provides a specification of the execution containers, their resource configuration (i.e., CPU, hard disk) and network connection while the allocation model specifies the deployment of the components to the execution containers. The usage model specifies the interaction of the users with the system. It describes user groups, the user behavior and usage intensity. The usage model shows a UML activity diagram like syntax with predecessor/successor relations between the model elements. Quality-relevant properties, like resource demands of actions and processing rates of resources, are part of the PCM models.

The iObserve megamodel is divided into four sections defined by two dimensions — one for development vs. operations, and one for model level vs. implementation level.

On the development side, the megamodel shows the architectural model that describes the software application to be developed. Moreover, the megamodel depicts the combination of the software application with a model-driven monitoring approach (Jung et al., 2013). The model-driven monitoring approach comprises the languages for specifying the monitoring. According to Jung et al. (2013), two languages have been developed to specify different characteristics of the monitoring. The instrumentation record language (IRL) defines the data structures used for monitoring in a record type model. This record type specification can be reused for different applications. The instrumentation aspect language (IAL) specifies the collection of data and the probe placement in an instrumentation model. This is specific to the application to be observed. Observing the five change scenarios in Section 2 requires the specification of probes for service invocations, component deployment and component undeployment. The architectural model, the record type model and the instrumentation model are used for generating source code of the application (i.e., implementation artifacts) and the corresponding monitoring probes (record type implementation) that are woven in the application source code by aspect-oriented programming (aspect implementation). Details on the model-driven monitoring approach are described in Jung et al. (2013).

On the operation side, the architectural model is updated by monitoring data which results from observing the running application to create the architectural runtime model. The architectural runtime model reflects the current state of the running application and can be applied for analysis to identify upcoming quality flaws. For example, performance bottlenecks may occur in case of increased usage intensity and limited processing capacity in the given service offering of the current cloud provider (Heinrich, 2016). Once a quality flaw has been identified, candidate architectural models are generated to solve the flaw based on an adaptation planning approach (Pöppke, 2017). The adaptation planning approach builds upon the design space exploration approach PerOpteryx (Koziolek et al., 2011) to find appropriate adaptation candidates for given degrees of freedom in the architectural model. Deployment of components of the application to execution containers is an example of a degree of freedom. If an appropriate candidate architectural model has been found, the differences between the candidate model and the architectural runtime model are identified and specified in an adaptation model. In the example, the adaptation model will exhibit a change in component deployment from one execution container to another (for instance another data center or another cloud provider) to provide additional processing capacity. Based on the adaptation model, adaptation scripts are generated to automatically adapt the application. Details on the adaptation planning approach are described in Pöppke (2017). If no appropriate candidate model can be found or if adaptation cannot be executed automatically, human intervention is triggered.

4.2. Runtime architecture correspondence

The Runtime Architecture Correspondence Model (RAC) is depicted as the central element in the megamodel in Fig. 2 because it is essential for the use of an architectural model during development and operation. Monitoring results in observations of events with respect to source code classes like the invocation of a service of a class or the deployment of a class. The RAC bridges the divergent levels of abstraction between the monitoring outcome on implementation level and component-based architectural models. More precisely, the RAC contains the correspondence between a software component and the source code classes implementing the component. By correspondence we denote the knowledge about which class (or implementation artifact in general) originated from which component (or architectural model element in general). This knowledge is essential to update the architectural model based on observations on implementation level. Furthermore, this knowledge is required for weaving the monitoring probes into the application source code and for generating adaptation scripts.

Reproducing knowledge about a component-based architecture from source code and monitoring outcome, respectively, is a non-trivial task. Software applications often comprise hundreds or thousands of source code classes with many different dependencies more or less explicit. Finding appropriate partitions of such complex applications requires the usage of heuristics (Sadou et al., 2011). The related approaches discussed in Section 6 apply heuristics to identify software components, their boundaries as well as required and provided interfaces. Applying these heuristics results in more or less adequate architectural models which, however, often differ from the initial architectural model created during development, for example in terms of the level of abstraction.

For this reason, iObserve does not extract the architectural model from scratch but updates an existing development model based on monitoring data. To do so, we require aforementioned correspondences. Specifying the correspondences between elements of different models, often on different levels of abstraction, is addressed in several approaches in literature (e.g., Kramer et al., 2015; Hamlaoui et al., 2014; Yie et al., 2012). Typically, correspondence rules are specified to describe the complex relation between model elements. Due to the component-based architecture underlying the iObserve approach, there is a one-to-many (1:n) mapping between an element of the architectural model and the artifacts implementing that model element. Consequently, we can specify the modeling language to create the RAC by the metamodel depicted in Fig. 3. We use the UML class diagram notation in the figure as metamodels are commonly depicted as UML class diagrams. To avoid misunderstandings, it is important to note at this point that the RAC metamodel in Fig. 3 does not shows the actual architectural model and implementation artifacts depicted in Fig. 2. Instead, it shows metaclasses that represent them and refer to the actual architectural model elements and implementation artifacts as described in the following.

The correspondence model metaclass is the model root that contains the correspondence metaclass. Furthermore, it contains a metaclass to represent the architectural model and a metaclass to represent a set of implementation artifacts. The metaclass

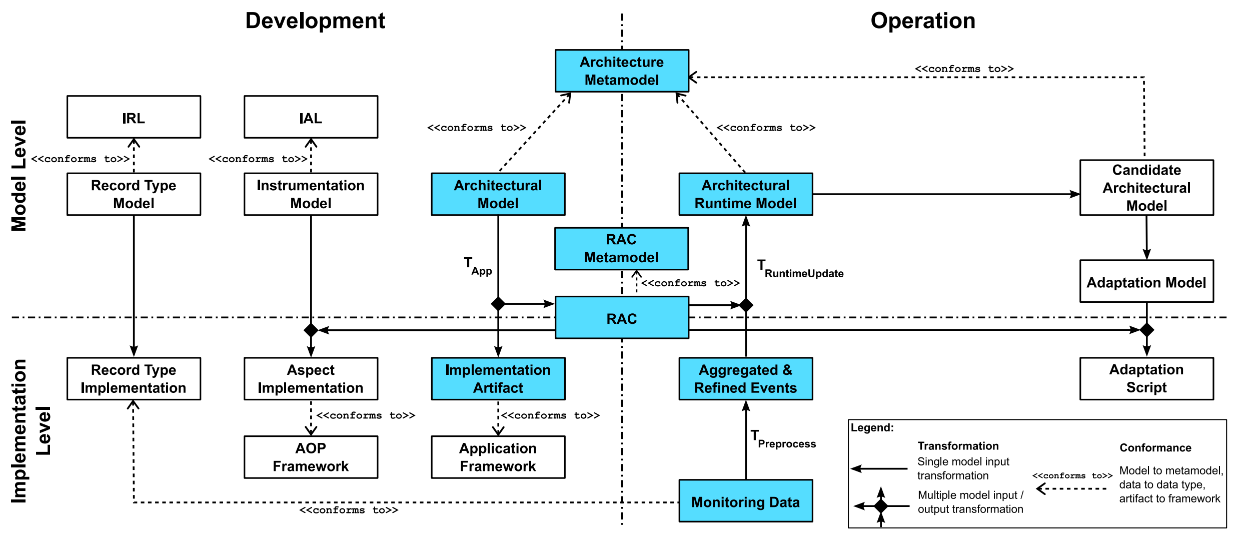


Fig. 2. Overview of the iObserve megamodel.

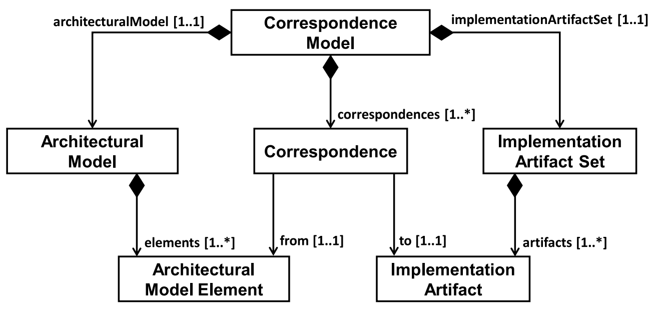


Fig. 3. Runtime architecture correspondence metamodel.

architectural model element refers to an element of the actual architectural model in Fig. 2. The metaclass implementation artifact refers to an actual implementation artifact in Fig. 2. Consequently, the correspondence metaclass specifies the relation between one element of the architectural model and one implementation artifact. If an element of the architectural model is implemented by several implementation artifacts, several correspondences are specified in the RAC.

During the initial application development or evolutionary changes by the developer, the source code stubs of the application are generated from the architectural model by the transformation T_{App} in Fig. 2. For this purpose, we use the tool ProtoCom (Becker et al., 2008) which is part of the Palladio Bench (Reussner et al., 2016). Based on the architectural model, ProtoCom generates Java interfaces and classes, method signatures within the classes and calls to external services as well as deployment scripts. While generating the code, the correspondence information between a generated class (represented by the implementation artifact metaclass in Fig. 3) and the component of the architectural model (represented by the architectural model element metaclass in Fig. 3) is automatically stored in the RAC by an extension of the ProtoCom code generator.

During operation, the software application and the entire cloud-based system faces various changes (cf. Section 2). These changes require the initial architectural model to be updated to continuously reflect the current state of the running system. Updating the architectural model based on observations on implementation level must not deviate its component-based fashion and, thus, its usefulness for humans in long-term evolution and operator-in-the-loop adaptation. 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 metamodel, and (b) the correspondence between an element of the architectural model and the implementation artifacts is recorded in the RAC while code generation during development and (c) restored while updating the component-based architectural runtime model by monitoring data related to the implementation artifacts. The level of abstraction of the initial model does not affect the correspondence specification in the RAC. Therefore, in analogy to existing component models, we do not predetermine the abstraction level used in the architectural model. Consequently, due to the correspondence between the architectural model and the implementation artifacts specified in the RAC, the abstraction level of the model cannot deviate from one update to another.

4.3. Transformation pipeline

After the application has been deployed in an execution environment, it is observed using aforementioned monitoring approach. As depicted in Fig. 2, iObserve filters and aggregates the monitoring data regarding the five change scenarios in Section 2 using the transformation $T_{Preprocess}$. Subsequently, iObserve automatically relates the monitoring data to elements of the architectural model based on the correspondence relations specified in the RAC and uses the aggregated data to drive runtime model update by the transformation $T_{RuntimeUpdate}$. The transformations are implemented based on the high throughput and parallelizing pipe and filter framework Teetime (Wulf et al., 2016) to automatically schedule and execute the transformations.

Fig. 4 illustrates the inner structure of the $T_{Preprocess}$ transformation. The notation for this figure and the following figures in this section is as described hereafter. Rectangles represent models. Solid lines depict transformations between the models. Note, monitoring data and events are instances of a metamodel according to our model-driven monitoring approach (Jung et al., 2013) and thus considered models in iObserve. Starting with monitoring data on the left side, F_{Trace} filters service call entry and exit events used to determine call traces through the software application. Each service call event comprises the point in time

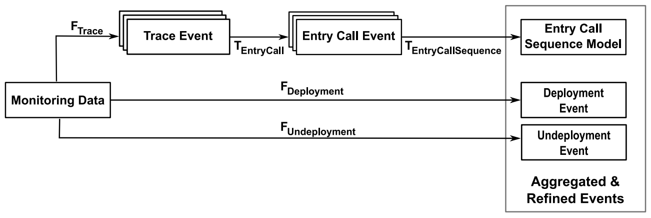


Fig. 4. Inner structure of the $T_{Preprocess}$ transformation.

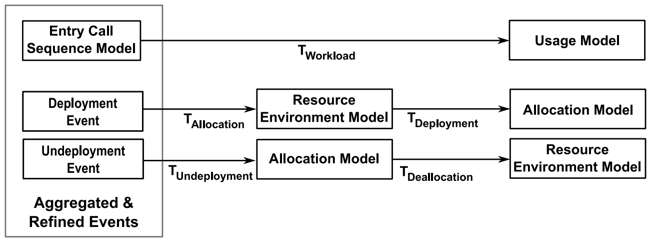


Fig. 5. Inner structure of the $T_{RuntimeUpdate}$ transformation.

the event occurred, session information, the name of the invoked service, a counter that indicates whether it is an application entry-level operation, and the full qualified name of the class that implements the service. The $T_{EntryCall}$ transformation listens to that event stream to identify events of application entry-level operations and create entry call events. Based on the session information, the transformation $T_{EntryCallSequence}$ aggregates all entry calls of each observed user session and combines them in a graph-based entry call sequence model.

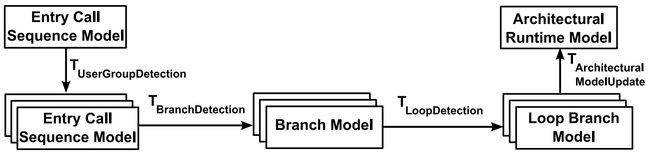
Component deployments and undeployments are filtered by $F_{Deployment}$ and $F_{Undeployment}$, respectively, and can directly be mapped to the corresponding events as there is no aggregation required.

Fig. 5 depicts the inner structure of the $T_{RuntimeUpdate}$ transformation. Based on the aggregated and refined events provided by $T_{Preprocess}$, the transformation $T_{RuntimeUpdate}$ comprises transformations for updating the architectural runtime model to reflect changes observed in the running system. The RAC is used in these transformations to identify the elements of the architectural runtime model corresponding to the observed implementation artifacts. For the sake of brevity the RAC is not depicted and only the most relevant transformations are visualized in the figure.

The transformation $T_{Workload}$ updates the user behavior and usage intensity (C1) in the PCM usage model. Updates in user behavior and usage intensity are further discussed in Section 4.4.

Changes in deployment (C2 to C4) are represented in the PCM allocation model by the transformations $T_{Deployment}$ and $T_{Undeployment}$. Each deployment event contains information about the deployed implementation artifact (i.e., unique identifier and class name) and the execution container the artifact is deployed to. Also each undeployment event contains information about the implementation artifact and the execution container the artifact is removed from. The components on architecture level corresponding to the implementation artifacts can be identified based on the correspondence relations specified in the RAC. Consequently, the PCM allocation model can easily be updated by $T_{Deployment}$ and $T_{Undeployment}$ for given deployment and undeployment events by changing the deployment of components to the corresponding execution containers.

The execution containers in the PCM resource environment model (C5/C6) are updated by the transformations $T_{Allocation}$ and $T_{Deallocation}$. Deployment and undeployment events are employed to update the PCM resource environment model. If a deployment event shows a deployment of a component (its implementation artifacts to be specific) to an execution container not yet contained in the resource environment model, the model is updated by $T_{Allocation}$ for adding the new execution container. Then, the transformation $T_{Deployment}$ is invoked for adding the component deployment to the execution container. The new execution container, however, does not yet have a network connection to the other execution containers in the resource environment. The network connection is derived based on observed service calls. If a service of a component deployed on another execution container calls a service of the component deployed on the new execution container, we can assume a network connection between the two execution containers. If an undeployment event removes the last component deployment from an execution container in the model, the execution container and its network connections are removed from the resource environment model by the transformation $T_{Deallocation}$.



Entry Call Sequence Model
$T_{UserGroupDetection}$
Entry Call Sequence Model
$T_{BranchDetection}$
Branch Model
$T_{LoopDetection}$
Loop Branch Model
$T_{Architectural ModelUpdate}$
Architectural Runtime Model

Fig. 6. Transformation pipeline for modeling of user behavior within $T_{Workload}$.

4.4. Behavior modeling

For adequately modeling complex user behavior in $T_{Workload}$ it is not sufficient to only aggregate service calls to sequences. Control flow elements like branches and loops must be detected and the single call sequences must be bundled to clusters, each representing the behavior of a certain user group. Moreover, usage intensities must be determined based on time stamps associated to the call sequences. In the following, we describe the behavior modeling approach of iObserve.

Starting with the entry call sequence model provided by $T_{Preprocess}$ we first identify different **user groups** within the observed user sessions. Once the several user groups are distinguished the group-specific behavior can be modeled individually. This enables analyses, for instance, on the impact of changes in a user group's specific workload on the application performance.

A user group is defined by a set of user sessions that exhibits a similar behavior pattern reflected by the services invoked. This means the types and count of service calls must be similar within a user group. The identification of different user groups is handled by the transformation $T_{UserGroupDetection}$ depicted in Fig. 6. For clustering the user sessions into groups a distance measure is required for determining the similarity between the single user sessions. The count of each distinct service call is used for the estimation of similarity. The user sessions are clustered into distinct user groups based on clustering principles we adapted from Menascé et al. (1999) and van Hoorn et al. (2014). We modified these principles as follows.

The number of possible user groups is input to these existing approaches. However, the number of user groups is often unclear when estimating clusters from scratch. This threatens accuracy of existing approaches. In contrast, we do not create the user groups from scratch but build upon the iObserve megamodel to leverage knowledge on expected user groups from the PCM usage model created during development for cluster estimation.

We apply the X-Means clustering algorithm (Pelleg and Moore, 2000) which is an extension of the established K-Means algorithm (Alsabti, 1998) for efficient cluster estimation. The clustering algorithm assigns each user session to exactly one user group based on the number of matching service calls. In other words, user sessions with a similar number of matching service calls are assigned to the same user group. The clustering result is a set of entry call sequence models, each representing a particular user group and containing exclusively the associated user sessions.

Further, the **usage intensity** for each of the identified user groups is calculated based on the contained user sessions. Basically, two types of workload can be distinguished — open and closed workload (Reussner et al., 2016). An open workload is specified by a certain inter-arrival time between users (represented by user sessions) requesting the system. A closed workload is specified by a certain user population on the system. Whereas approaches like (Langhammer et al., 2016) do not support the definition of usage intensity (usage intensity must be added manually after the model has been extracted) and WESSBAS (van Hoorn et al., 2014) is limited to closed workload specifications, our approach supports both types of workload.

For deriving an open workload we calculate the inter-arrival time distribution from the given user sessions within a group. Therefore, the distances between the entry time points of the timely ordered user sessions are used. For deriving a closed workload the mean number of concurrent user sessions on the system is determined. First, time intervals are calculated from the points in time each user session enters and exits the system. These intervals are used to divide the observed time into time frames. Where for each time frame the number of active users can be counted.

For identifying **branches** in the user behavior, the transformation $T_{BranchDetection}$ aggregates the call sequences within a user group to construct a specific branch model for each user group. Equal service calls within a sequence are merged and alternative service calls are represented as branches. Each branch transition contains a call sequence and a transition probability calculated from the ratio of user sessions assigned to the specific transition to the overall number of user sessions of the branch. Consequently, the branch model reflects a call sequence including branches. Existing approaches (cf. Menascé et al., 1999; van Hoorn et al., 2014) apply transition frequency matrices that state the transition probabilities between the single service calls for each user group. This is a simplified representation of the user groups' behavior as it only states the probabilities of which a next service is called. In contrast, iObserve takes each single service call into account (instead of transition probabilities) to reflect precisely the user behavior actually observed during operation (Peter, 2016).

The transformation $T_{LoopDetection}$ identifies **loops** within call sequences. For each sequence, the repeated occurrence of service calls is identified which may comprise a single or multiple service calls. $T_{LoopDetection}$ provides for each user group a loop branch model. Iterations are represented by loop elements annotated with a loop count that specifies the number of iterations. As existing approaches (cf. Menascé et al., 1999; van Hoorn et al., 2014) rely on transition probabilities, they do not support the detection of loops. They can only model transitions of a single service call to itself. iObserve in contrast is able to detect even complex user behavior.

The transformation $T_{ArchitecturalModelUpdate}$ produces a PCM usage model from the loop branch models. For each user group a separate usage scenario is created within the usage model that reflects the obtained entry calls, branches, loops and the corresponding usage intensity.

4.5. Discussion on applicability

This section refers to the applicability of the proposed modeling concepts by discussing first their dependency to tools and languages mentioned in the previous sections and second the effort for performing manual steps of the iObserve approach.

We use the tools Palladio Bench, ProtoCom and Kieker as well as the programming language Java for implementing and illustrating iObserve. However, there is no conceptual limitation to these tools and languages. iObserve can be applied based on an arbitrary modeling language that is able to reflect the changes during operation described in Section 2. This means the language must be able to express at least the application's usage intensity and user behavior, software components and their interfaces, execution containers and the deployment of components to execution containers. An arbitrary code generator can be applied that is able to generate source code in an arbitrary object-oriented language from the component-based model. Of course, the code generator must be extended to fill the RAC when generating code. The running system can be observed using an arbitrary monitoring framework that is able to observe the changes during operation. In Kunz et al. (2017), we proposed a platform to use an arbitrary monitoring framework for iObserve.

Most steps of iObserve are automated. Nevertheless, some steps need to be performed by humans. For these steps we discuss the effort in the following. The developer creates an initial architectural model and updates the architectural model due to evolutionary changes. The effort for creating or updating the architectural model depends on the extent of the system and changes to be modeled and the level of abstraction of the model as these factors determine the amount of information to be represented in the architectural model. As mentioned before, we do not make any specification regarding the level of abstraction of the model.

Moreover, the developer may specify and place monitoring probes using the model-driven monitoring approach (Jung et al., 2013). iObserve addresses the most common changes during operation (see Section 2) of cloud-based software applications discussed in literature (Heinrich, 2016). For these changes the record type specification can be reused for different architectural models without additional effort. Merely the probe placement and data collection may be adapted. Observing other changes would require to update the models and transformations of the iObserve megamodel. The effort of this heavily depends on the new changes to be observed and their representation in the models.

The developer manually extends the source code stubs generated from the architectural model with details that cannot be generated automatically. To be more specific, the developer implements the method bodies in the stub classes. This is because the PCM contains the specification of external calls and quality properties of internal actions, like resource demands, but there are no implementation details specified in the PCM. The PCM purposely abstracts from implementation details as they may not be available in early design, on the one hand, and on the other hand to force the modeler to abstract from details not relevant to quality properties. This way, the PCM prevents the modeler from "programming" (i.e. specifying information on implementation level) in the PCM as this is part of the following development phases and thus keeps track of the model. We believe this is useful also for other component-based modeling languages. The effort for manual extension obviously depends on the number and extent of method bodies to be implemented.

In order to create a running system, the operator deploys the completed application in an execution environment. This can be done manually or in automated fashion but is not specific to the iObserve approach.

5. Evaluation

The architectural runtime models of the iObserve approach must reflect the changes during operation described in Section 2. Therefore, we designed experiments based on an established community case study described in this section. The experiments are designed according to the changes along the following goals.

Goal 1: We evaluate the **accuracy** of the architectural runtime models of iObserve for reflecting changes in workload characterization, component deployment and allocation of execution containers.

Runtime modeling approaches face specific requirements towards their reaction times according to Cheng et al. (2009). For instance, the runtime model in our work has to be updated timely as this is required for running analyses and mitigative actions close to the observation of changes during operation. Consequently, in

Goal 2: We examine the **scalability** of the transformation pipeline for updating the architectural runtime models to reflect changes during operation.

Note, the transformation T_{App} is not evaluated in this paper as it is an extension of ProtoCom which has already been evaluation in Becker et al. (2008).

5.1. A sample system for evaluation

The evaluation of iObserve builds upon an established case study from the software architecture modeling and analysis community — the Common Component Modeling Example (CoCoME) (Herold et al., 2008; 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. In an evolution scenario (Heinrich et al., 2016), CoCoME has been extended by an online shop where customers can buy products online. The detailed design and implementation of CoCoME is described in Heinrich et al. (2016) and the source code is available for download.[2] CoCoME is deployed in a cloud-based data center.

An overview of user groups and use cases of CoCoME relevant for the evaluation of iObserve is given in Fig. 7. The use cases comprise the services of CoCoME. CoCoME provides 13 different services. The service invocations will be observed and processed by the iObserve transformation pipeline for evaluation. All user groups involved in the system will invoke the login and logout services. A service typically invoked by a Customer is the sale service (part of the use case ProcessOnlineSale). A Stock Manager may invoke the check delivery and update stock services (part of the use case ReceiveOrderedProducts). A Store Manager may invoke the change price and order product services (part of the use case ChangePrice and OrderProducts, respectively) (Herold et al., 2008).

As a trading system, CoCoME may face variations in usage intensity from time to time. For example, advertisement campaigns of the supermarket chain may lead to increased number of sales and thus changes in the application's workload characterization (C1). Components of the CoCoME application may be migrated (C2) from one data center to another, replicated (C3) or dereplicated (C4). Furthermore, data centers for deploying the components of CoCoME may become available (C5) or disappear (C6).

2 https://github.com/cocome-community-case-study.

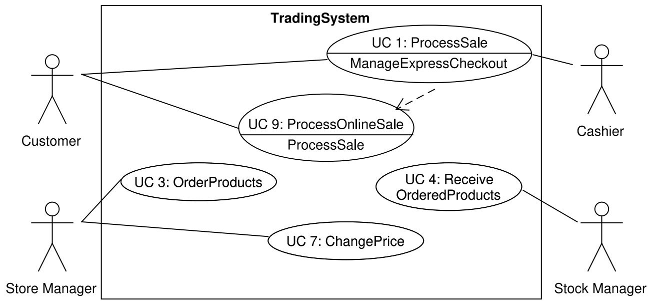


Fig. 7. Excerpt of the use cases of CoCoME.

5.2. Research questions and metrics

Following the goal-question-metric approach (Basili et al., 1994) we derive research questions and associated metrics from the evaluation goals. These are described per goal hereafter.

5.2.1. Accuracy

We derive two research questions from Goal 1.

RQ1.1: How accurate does iObserve reflect the changes during operation (see Section 2) in the architectural runtime model?

RQ1.2: How accurate does iObserve reflect user groups in the architectural runtime model?

For answering RQ1.1, we first evaluate changes in workload characterization by conducting a series of experiments. For each experiment, a manually specified PCM usage model serves as a reference model. In the reference model we specify the changes in workload characterization to be evaluated in form of a PCM usage model. We use a script to automatically generate user sessions that comprise the user behavior and usage intensity that exactly corresponds to the specified reference model. The user sessions are represented in form of aggregated and refined events in an entry call sequence model. iObserve is applied to update the current usage model of the application based on the generated user sessions. We evaluate whether iObserve adequately reflects workload changes by comparing the updated usage model to the reference model. A detailed experiment design is described in Section 5.3.

Second, for evaluating changes in deployment and allocation we again conduct a series of experiments. We manually specify reference models and generate deployment and undeployment events which are then input to the iObserve transformation pipeline for updating the current architectural model of the application. For each change in component deployment and each (de)allocation of an execution container we examine whether it is correctly reflected in the architectural runtime model by comparing the updated models to the reference models.

In the experiments on **user behavior**, we compare a reference PCM usage model to the PCM usage model generated by iObserve. Therefore, we apply the following set-based metrics to models. Two sets are considered equal if their contained elements are equal. The Jaccard Coefficient (JC) (Levandowsky and Winter, 1971) determines the similarity of two sets A and B by comparing their elements. The more elements of the sets are equal, the more similar are the sets to each other. We choose the JC as it is a simple yet powerful metric to compare two sets for equality. The JC ranges from 0.0 to 1.0. A value of 1.0 indicates the sets contain only equal elements. A value of 0.0 indicates the sets do not share any element.

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

We consider a PCM usage model as a set of elements. The elements of the reference model are compared to the elements of the generated model. We denote a JC value of 1.0 as an exact representation. Any JC value less than 1.0 is considered as a corrupt representation.

Even if only equal elements are contained in both models, they may be interrelated to each other in a different way. We need to consider this in the experiments for the representation of user behavior in a PCM usage model. In order to evaluate the representation of user behavior we apply the Spearman's Rank Correlation Coefficient (SRCC) (Wayne, 1990). The SRCC is a measure for the similarity of the ordering of two lists L_1 and L_2. We assume the set of all elements E contained in a list are equal for both lists (see JC). Let $rank(L, e)$ denote the index of the element e in the List L and $n = |E|$ the total number of elements. Then, SRCC is defined as:

$$SRCC(L_1, L_2) = 1 - \frac{6\sum_{e\in E}(rank(L_1, e) - rank(L_2, e))^2}{n(n^2 - 1)}$$

It is important to note at this point that in general the order of elements in models may differ although they are interrelated the same way. For example, model elements referred by other elements within the model may be placed at different positions. However, we construct our reference PCM usage models in a way that the order of the elements conforms to their predecessor and successor relationship (also for nested elements). Consequently, we can assume in the experiments that given two equal PCM usage models, the order of elements within the model is equal, too.

We choose the SRCC because of its intuitiveness and simplicity to compute compared to other similar metrics, such as the Kendall Rank Correlation Coefficient (Kendall, 1938). A SRCC value of 1.0 reflects the same order in both lists. A SRCC value less than 1.0 reflects less similarity in the ordering of both lists.

We consider a PCM usage model as a list of model elements. The SRCC is used to investigate the similarity of the order in the reference model and the generated model. The prerequisite same number of model elements has already been checked by the JC before. Again, we consider a SRCC of 1.0 as an exact representation and any value less than 1.0 as a corrupt representation.

For evaluating the accuracy of the **usage intensity** we calculate the Relative Measurement Error (RME) (Abramowitz, 1974) between the reference usage intensity and the generated usage intensity. The RME is determined by the true value x_t and the measured value x_m. In our evaluation, x_m is not measured but generated by iObserve. A RME value close to 0.0 indicates low error.

$$RME = \frac{x_m - x_t}{x_t}$$

In the experiments on changes in **deployment and allocation**, we use the JC for comparing the models generated from deployment and undeployment events to the reference models. Models involved in these experiments are the PCM allocation model and the PCM resource environment model, respectively. We do not apply the SRCC for the experiments on changes in deployment and allocation as, in contrast to the PCM usage model, the order is not relevant for elements of the allocation model and the resource environment model. In other words, there is no defined predecessor/successor relationship between the elements of the resource environment. Two models are equal merely if they contain equal execution containers, equal components and equal deployments of the components to the execution containers.

For answering RQ1.2, we compare **user groups** in the reference usage model to those in the model updated by iObserve. The iObserve transformation pipeline applies a clustering algorithm to distinguish monitored user sessions into specific user groups. In our experiment, each user session is labeled with membership information of a certain user group. We apply the following metrics for comparison.

The percentage of misclassified user sessions (MC) (van Hoorn et al., 2014) indicates the accuracy of the clustering. This metric has already been used in related studies (e.g., WESSBAS van Hoorn et al., 2014). For each of the resulting clusters k, the number of misclassified user sessions (MCUS) is summarized. The sum is divided by the overall number of user sessions (US). Thus, the MC reflects the percentage of the user sessions that were assigned to wrong clusters.

$$MC = \frac{\sum_{i=1}^{k}|MCUS_i|}{|US|}$$

In addition, we apply the metric Sum of the Squared Error (SSE) (Pelleg and Moore, 2000) to investigate the compactness of the clusters. The SSE is commonly used for evaluating the accuracy of a clustering algorithm. For each of the k clusters Cl_i, the squared distances between each data point $p \in Cl_i$ and its centroid c_i is summarized.

$$SSE = \sum_{i=1}^{k}\sum_{p\in Cl_i}(p - c_i)^2$$

The closer the data points to each other (i.e., the lower the SSE), the more similar are the data points. We apply the SSE to evaluate the similarity of user sessions within a certain cluster. High SSE values indicate the cluster should be split.

5.2.2. *Scalability*

While examining the scalability of iObserve two dimensions of scalability must be distinguished. The first dimension refers to an increased number of users calling the same service within a given time frame. This means, although the observed user behavior remains the same, the number of monitored events may increase in case of increasing number of users that call the application (cf., advertisement campaign). In this dimension, the size of the resulting models is not affected while the number of monitoring events to be processed increases. The second dimension refers to an increased number of services called by a constant number of users. This means, although the number of users of the application remains constant, the transformations may need to handle increased number of events in case the size of user behavior increases (i.e. the single users call additional services). In this dimension, the size of the resulting models increases while the number of monitoring events to be processed increases. Consequently, we derive the following research questions from Goal 2 for analyzing the scalability of iObserve.

RQ2.1: How does iObserve scale with increasing number of users calling the same service?

RQ2.2: How does iObserve scale with increasing number of services called by the same user?

We measure the wall-clock time required for executing the transformations while increasing the number of events along the two dimensions. We denote this time as execution time in the following.

5.3. *Experiment design*

The design of experiments to investigate iObserve with respect to the evaluation goals are described in the following.

5.3.1. Accuracy

In iObserve, two transformation stages – $T_{Preprocess}$ and $T_{RuntimeUpdate}$ – are used to continuously preprocess monitoring events and subsequently update the architectural runtime model (cf. Fig. 2). For evaluating the accuracy of iObserve we apply preprocessed events generated from manually specified reference models instead of using monitoring data. This is necessary to exclude influence factors of load drivers and the monitoring framework. Furthermore, the generated events allow us specifying any combination of model elements we want to investigate in the experiments. As we do not use monitoring data, only the transformation $T_{RuntimeUpdate}$ is evaluated in the experiments on accuracy. Preprocessing of monitoring events is not required.

For evaluating the accuracy of representing changes in workload characterization we conduct several experiments that refer to different user behaviors and workload types. For each experiment a reference model is created and user sessions are constructed that correspond to the reference model. The user sessions are input to the iObserve transformation pipeline which updates the PCM usage model. Each experiment is repeated in several experiment runs. For each experiment run the updated usage model is compared to the reference model using aforementioned metrics.

We repeat each experiment in 500 runs to ensure a sufficient number of samples and thus achieve a certain reliability of the results. Each experiment run comprises a random number of user sessions in the range of 1 to 200 due to the following reasons: (a) We apply the random number of user session to evaluate that the transformation is not affected by the number of considered user sessions. (b) The random number of user sessions allows for evaluating the representation of usage intensity. Both types of workload, open and closed workload, depend on the number of user sessions and the entry and exit time points of the user sessions. A random number of user sessions and random entry and exit time points of the user sessions ensure that varying workload is created. The range of 1 to 200 user session ensures a sufficient variance in the number of user sessions to be processed.

The **user behavior** to be evaluated in the experiments comprises the PCM model elements system entry calls (i.e. application entry-level service calls), branches, loops, as well as related properties like probabilities of branch transitions and loop iterations. Each experiment comprises 500 runs as argued before. For each experiment run the length of a call sequence (1–5 service calls), the number of transitions of a branch (2–5 branch transitions), the transition probabilities of a branch (the sum of the transition probabilities has to yield 1), and the loop count of a loop (2–5 iterations), respectively, are set randomly within the given ranges. Each experiment run is configured randomly to evaluate the model accuracy is not affected by the number of model elements and their properties. Aforementioned ranges are chosen for two reasons: (a) the number of possible configurations is kept within reasonable limits and thus each configuration is repeatedly evaluated during the 500 experiment runs. (b) a larger range of model elements would not increase the evaluation significance as per type of model element the same transformation is executed. Consequently, increasing the range of model elements would merely result in increased repetitions of the same transformations with the same input.

For the experiments to evaluate the representation of user behavior we combine the PCM usage model elements in various reasonable combinations. This results in eight experiments — one experiment for each of the following bullet points:

i. call sequence,
ii. branch,
iii. loop,
iv. overlapping loops,
v. nested branch
vi. nested loop,
vii. loop within branch, and
viii. branch within loop.

The experiments are configured randomly but within the aforementioned ranges for the model elements. For evaluating the accuracy of iObserve to handle call sequences, a reference model and user sessions are created that represent call sequences of random length. The accuracy of representing branches is evaluated using a reference model and user sessions that represent a branch with a random number of branch transitions and a random specification of the branch transition probabilities (sum of the transition probabilities is always 1). Each user session in this experiment contains a call sequence after the branch transition that is common to all user sessions. This is needed to evaluate the merging of the control flow after the branch. For evaluating loops a reference model and user sessions are created that represent repeated call sequences with a random number of repetitions. Each call sequence to be represented by the loop element consists of a random number of service calls. Note, two call sequences where each represents a loop may overlap. In this case only one call sequence can be transformed to a loop element. In order to construct compact models we decide to transform the call sequence that comprises the most service calls to the loop element. In the experiment on overlapping loops either additional service calls of the first loop or additional service calls of the second loop are added. Thus, the number of service calls of one sequence is increased while the number of the other sequence remains the same. The accuracy of representing nested branches is evaluated by a reference model and users sessions that reflect nested alternative call sequences. Again, the number of branch transitions and transition probabilities are set randomly. The accuracy of representing nested loops is evaluated by a reference model and user sessions that comprise a call sequence in a random number of repetitions. The repeated call sequence again contains a call sequence repeated randomly. Representing a loop within a branch is evaluated by a reference model and users sessions that reflect alternative call sequences while each sequence contains repeated service calls. Representing a branch within a loop is evaluated by a reference model and user sessions that reflect alternative sequences of repeated service calls. The number of service calls in the sequence and the number of repetitions is chosen randomly.

The accuracy of **usage intensity** is evaluated by varying entry and exit times of the created user sessions. The representation of an open workload is evaluated by a random definition of a time distance between the user sessions' entry times. The time distance constitutes the mean arrival time of the created user sessions. For the representation of a closed workload the entry and exit times of the random number of user sessions are varied. Thus, a random number of concurrent user sessions is created which represent the population count of a closed workload specification. For the experiment on the accuracy of usage intensity the user sessions comprise a call sequence like in (i) of the user behavior experiments. This is because only the entry and exit times are considered in this experiment, not the user behavior in between.

Next, the experiment design for evaluating the accuracy of the transformations for changes in **deployment and allocation** is described. We create deployment and undeployment events for a given number of execution containers to evaluate the update of the architectural runtime model with respect to changes in deployment and allocation. In our experiments we use five execution containers (denoted as nodes n1 to n5) and one component (denoted as c). The number of events in the experiment is sufficient as an increased number would result in repeated execution of the same transformations with similar input. The generated events reflect the following deployment changes.

For evaluating component migration the generated events reflect a sequence of deployments and undeployments. We start with the component c deployed on the first execution container n1. The component is stepwise migrated to the next execution container until the component has been deployed on n5. This means the component is undeployed on the current execution container and deployed on the next execution container. For each migration we evaluate whether it is correctly reflected in the architectural model by the transformations $T_{Deployment}$ and $T_{Undeployment}$.

For evaluating component replication we generate a similar sequence of deployment events. In contrast to aforementioned sequence, the component c is not undeployed but it is deployed to the next execution container until a replication is deployed on n5. Dereplication is evaluated the reverse way. We generate undeployment events starting with n5 until the replication on n2 has been undeployed. For each (de)replication we evaluate whether it is correctly reflected in the architectural model by the transformations $T_{Deployment}$ and $T_{Undeployment}$, respectively.

For evaluating the representation of allocation we again create a sequence of deployment events to the five execution containers. In contrast to aforementioned experiments, the execution containers are not yet contained in the PCM resource environment model. Therefore, not the handling of the deployment changes by $T_{Deployment}$ is in focus of this experiment but the creation of new execution containers in the PCM resource environment model by the transformation $T_{Allocation}$. For evaluating the representation of deallocation we create a sequence of undeployment events to the five execution containers. Not the handling of the deployment changes by $T_{Undeployment}$ is in focus of this experiment but we evaluate whether the execution containers are removed from the PCM resource environment model by the transformation $T_{Deallocation}$ once there is no longer a component deployed. For each (de)allocation we evaluate whether it is correctly reflected in the architectural model.

Further, we need to evaluate whether our clustering approach assigns user sessions of the same **user group** to the same cluster. Therefore, we model exemplary behaviors of three user groups – Customer (CR), Stock Manager (SKM) and Store Manager (SEM) – of the CoCoME trading system and its web shop extension (Heinrich et al., 2016). Fig. 8 depicts the behavior for each of the three user groups by service calls and transition probabilities to subsequent service calls in a UML activity diagram like syntax. The modeled behavior is the reference model for our experiments on clustering.

We construct the Store Manager user group in a way that it does not represent the best fitting cluster. Within the Store Manager user group, a subgroup of users that mainly change prices and another subgroup of users that mainly order products can be identified. Therefore, we apply the Store Manager user group to evaluate whether it is split by our clustering approach once a certain variance factor is applied for clustering.

In analogy to related studies (van Hoorn et al., 2014), we specify three different behavior mixes by alternately doubling the load (i.e. number of user sessions) of one of the three user groups. This setting allows for evaluating whether the behavior mix affects the clustering accuracy. The call sequence of each user session is randomly generated according to the reference model of the respective user group. Fig. 9 shows two exemplary call sequences for each user group.

We apply this setting in two different configurations to evaluate the variance parameter of the X-Means clustering algorithm. For both configurations the number of user groups is known from the existing PCM usage model. With the first configuration we

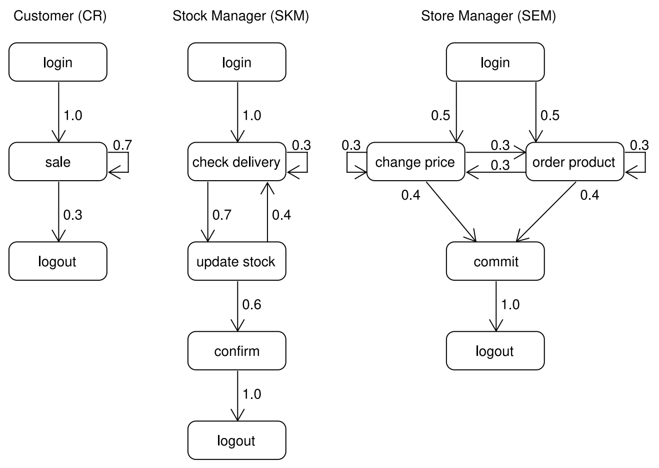


Fig. 8. Reference model for the clustering experiments.

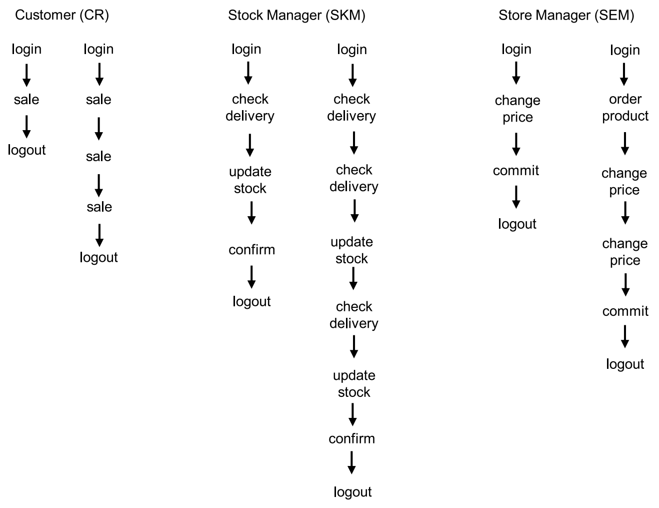


Fig. 9. Exemplary call sequences for the clustering experiments.

evaluate the accuracy of iObserve to identify predefined clusters. For the parametrization of the X-Means clustering algorithm the number of user groups is set to 3 and the variance of user groups is set to 0. With the second configuration we evaluate whether iObserve is able to identify better fitting clusters within a given range. A variance value of 10 is introduced. Consequently, the minimum number of user groups is set to 2 (2 is the minimum number of clusters) and the maximum number of user groups is set to 13. A variance value of 10 has been chosen to provide the clustering algorithm with a range of potential clusters that is wide enough to enable a good clustering. Changes in user groups are possible during operation when users start to behave in a way not expected before. This might be the case if new services are available, the user's preferences for invoking certain services change or services are not available any longer.

Each of the two configurations is executed for 8000 user sessions. Each user session is labeled with the user group it belongs to. The number of user sessions in the experiment is created according to the given user group mix. For each user session the user behavior is created randomly according to the reference model of the respective user group. 8000 user sessions guarantee each user group is represented by user session with high variance of user behaviors.

5.3.2. *Scalability*

In the scalability experiments, the two transformation stages $T_{Preprocess}$ and $T_{RuntimeUpdate}$ are used to preprocess monitoring events and subsequently update the architectural runtime model on that basis. Although the experiments include both stages – $T_{Preprocess}$ and $T_{RuntimeUpdate}$, the two stages are addressed separately to better locate potential scalability limits.

As simultaneous user interactions with the cloud-based application may lead to a large number of monitoring events, we put strong emphasis on examining the scalability of the user behavior update in our evaluation. Updating the user behavior for excessive system usage requires processing a variety of monitoring events and, therefore, is well suited to examine the scalability of the iObserve transformation pipeline in worst case.

Changes in deployment and allocation as described in Section 2 are less appropriate for analyzing the scalability of iObserve. First, they occur relatively seldom compared to user interactions which nearly occur permanently in large enterprise applications. Second, they do not require a comprehensive aggregation and preprocessing as can be identified directly by observable events for deployment and undeployment. Third, their impact on the updated model is relatively small compared to user behavior changes as only single elements (e.g., a deployment context) need to be updated.

In the experiments, we examine and discuss the scalability of iObserve while increasing the number of monitoring events to be processed in two dimensions as introduced before: (i) increased number of users calling the same service and (ii) increased number of services called by the same user.

For (i), we use the sale service of the CoCoME case study. We generate an increasing number of user sessions for which each user session contains exactly one call of the sale service. Starting with one (10^0) user session we increase the number of user sessions in orders of magnitudes until reaching 100 000 (10^5) user sessions. By increasing the number of user sessions the number of monitoring events to be processed is increased in the same extent. While increasing the number of user sessions we observe how this influences the execution time for updating an architectural runtime model. We chose an upper limit of 10^5 user sessions as this is expected to result in a reasonable large number of events to be processed.

For (ii), we randomly chose from the 13 services provided by CoCoME. We generate user sessions with an increasing number of service calls. Starting with one call we increase the number of calls within the user sessions in orders of magnitudes until reaching 10^5 calls. While increasing the number of calls we observe how this influences the execution time for updating an architectural runtime model. We chose an upper limit of 10^5 calls within a user session to be in line with the first experiment. In fact, we believe 10^5 calls is far beyond the number of calls typically contained in a user session.

Ten repetitions of the scalability experiments are sufficient to achieve reliable results. More precisely, we chose ten repetitions to reduce the impact of variation within a single experiment run on the overall results. The scalability experiments have been conducted on a virtual machine with Ubuntu Linux 16.04 64-bit edition. The machine is using four virtual CPUs with 3.41 GHz. The machine has 2 GB RAM and 100 GB disk space. As Java Runtime environment we used the OpenJDK 1.8 64-bit edition.

5.4. *Experiment results*

The results of the experiments to evaluate iObserve are described per evaluation goal in the following.

Table 1
Evaluation results of accuracy for representation of user behavior.

User Behavior	JC	SRCC
Call sequence	1.0	1.0
Branch	1.0	1.0
Loop	1.0	1.0
Overlaps	1.0	1.0
Nested branch	1.0	1.0
Nested loop	1.0	1.0
Loop within branch	1.0	1.0
Branch within loop	1.0	1.0

Table 2
Evaluation results of accuracy for representation of usage intensity.

Usage Intensity	RME
Open workload	0.0%
Closed workload	13.1%

5.4.1. *Accuracy results*

For answering RQ1.1, we first conducted the experiments on changes in workload characterization and applied the metrics JC and SRCC to interpret the experiment results. Table 1 gives an overview of the experiment results for the representation of **user behavior** in the usage model. For each of the experiments, both the JC and SRCC show the value 1.0. Thus, for each evaluated user behavior no single deviation between the reference model and the updated usage model was identified within 500 experiment runs.

Due to the experiment design we could demonstrate the accuracy of the transformations is not affected by influence factors like the number of user sessions or variances in number of loop counts and branch transitions. Consequently, the experiments show the transformations are able to accurately reflect observed changes in user behavior.

Table 2 shows the experiment results for **usage intensity**. For open workload the experiments resulted in a RME value of 0.0%. This means each open workload specified by a mean inter-arrival time in the usage model correctly represents the corresponding inter-arrival time in the user sessions created from the reference model. This is because the inter-arrival time can easily be calculated by $T_{Workload}$ based on the monitoring events. The RME of the closed workload specifications shows a value of 13.1%. This indicates the number of concurrent user sessions determined by iObserve deviates from the reference number of concurrent user session by 13.1% on average. In contrast to the open workload, the number of active users within a given time frame has to be approximated for the closed workload specifications. The deviation is caused by the approximation of the number of active users within a given time frame (see Section 4.4).

Next, we discuss the results of the experiments on changes in **deployment and allocation**. Starting with a base model we stepwise changed the deployment in each of the five experiments for migration, replication, dereplication, allocation and deallocation. The models created from the deployment and undeployment events are compared to the corresponding reference model using the JC. Table 3 shows the experiment results per step. Step means a change in deployment of the component for one of the five execution containers. The JC for all the changes in deployment and allocation in the experiments is 1.0. Consequently, the experiments demonstrate an exact representation of migration, replication, dereplication, allocation and deallocation in the models.

For answering RQ1.2 (the accuracy of the clustering to detect **user groups**), we use two different experiment configurations. In one configuration the number of user groups is fixed to evaluate the accuracy of iObserve to identify predefined clusters. In the

Table 3
Evaluation results of accuracy for representation of deployment and allocation.

Deployment/ Allocation	JC			
	Step 1	Step 2	Step 3	Step 4
Migration	1.0	1.0	1.0	1.0
Replication	1.0	1.0	1.0	1.0
Dereplication	1.0	1.0	1.0	1.0
Allocation	1.0	1.0	1.0	1.0
Deallocation	1.0	1.0	1.0	1.0

Table 4
Clustering results with variance value 0.

UG	UGM	Cl1	Cl2	Cl3	MC	SSE
CR	50%	0	0	4000		
SKM	25%	2000	0	0	0.0%	177.04
SEM	25%	0	2000	0		
CR	25%	0	2000	0		
SKM	50%	4000	0	0	0.0%	190.37
SEM	25%	0	0	2000		
CR	25%	0	0	2000		
SKM	25%	0	2000	0	0.0%	202.31
SEM	50%	4000	0	0		

other configuration we evaluate whether iObserve is able to identify better fitting clusters within the given range. A variance factor enables the clustering algorithm to determine a better fitting number of clusters (i.e. user groups).

Table 4 shows the results of the configuration using a fixed number of user groups (UG) and alternately doubling the load (i.e. number of user sessions) of one of the three user group mixes (UGM). Three clusters – Cl1, Cl2 and Cl3 – have been created. As seen in the table, the clustering algorithm correctly assigns each user session to the user group specified in the reference model. The misclassification rate for each load distribution in the user group mixes is 0.0%. The table further shows the number of user sessions of each user group does not affect the accuracy. In consequence, the evaluation results summarized in the table show the accuracy of iObserve to transform user session into predefined user groups.

The SSE values in Table 4 indicate the predefined user groups may not be the best fitting clusters, especially for double number of user sessions of SEM. Therefore, a variance factor is introduced to find better fitting clusters. Table 5 gives an overview of the results in case the variance value of user groups is set to 10. Four clusters have been created. For each load distribution the user sessions of user group SEM are divided into two separate subgroups. This can be explained by the high dissimilarity between the user sessions of user group SEM. A closer inspection of the resulting subgroups reveals the user sessions are divided into the one group that mainly changes prices and the another group that mainly orders products. Introducing the variance factor results in misclassification rates ranging from 6.01% to 13.77%. This is because some of the user sessions are no longer assigned to the clusters of user group SEM. By splitting the user group SEM, the SSE values in Table 5 decrease in comparison to Table 4. This indicates more compact clusters. In consequence, the evaluation results summarized in the table show iObserve is able to find better fitting clusters for given user sessions.

5.4.2. *Scalability results*

In the scalability experiments, we investigate the scalability of iObserve for the two dimensions (i) increased number of users calling the same service and (ii) increased number of services called by the same user by measuring the execution time of the transformations. For $T_{Preprocess}$ we analyzed the subtransformations $T_{EntryCall}$ and $T_{EntryCallSequence}$. For $T_{RuntimeUpdate}$, we analyzed only $T_{Workload}$ – there in detail $T_{UserGroupDetection}$, $T_{BranchDetection}$, $T_{LoopDetection}$, and $T_{ArchitecturalModelUpdate}$. The transformations $T_{Deployment}$, $T_{Undeployment}$, $T_{Allocation}$ and $T_{Deallocation}$ are irrelevant for the scalability experiments, due to the reasons discussed in Section 5.3.2. Each scalability dimension has been evaluated in a separate experiment. Fig. 10 shows the execution time measurements in milliseconds. Each experiment consists of ten repetitions to alleviate measurement errors. From these ten repetitions, we visualized the median in the result plots.

Fig. 10a shows the results of the experiment for increasing the number of users calling the same service for $T_{Preprocess}$. The execution time values of $T_{Preprocess}$ first increase sublinearly between 1 and around 100 users calling the same service and then increase in a linear manner between around 100 and 100 000 users calling the same service.

Fig. 10b shows the results of the experiment for increasing the number of users calling the same service for $T_{RuntimeUpdate}$. The execution time values of $T_{RuntimeUpdate}$ first increase sublinearly between 1 and around 100 users calling the same service and then increase in a slightly superlinear manner between around 100 and 100 000 users calling the same service.

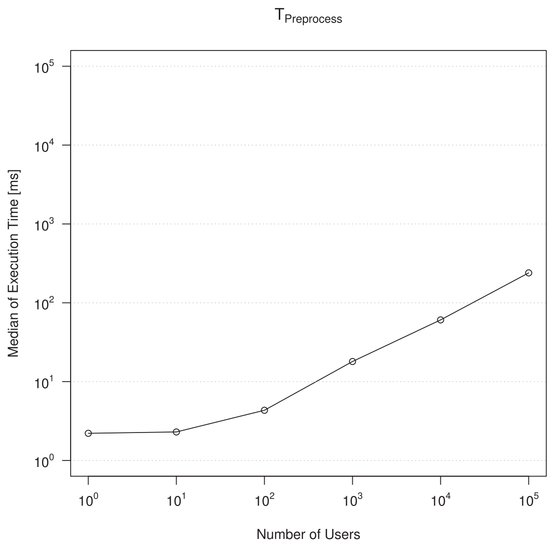
Fig. 10c shows the results of the experiment for increasing the number of services called by the same user for $T_{Preprocess}$. The execution time values of $T_{Preprocess}$ first increase sublinearly between 1 and around 100 services called by the same user and then increase in a linear manner between around 100 and 100 000 services called by the same user.

Fig. 10d shows the results of the experiment for increasing the number of services called by the same user for $T_{RuntimeUpdate}$. The execution time values first scale sublinearly between 1 and around 100 services called by the same user and then increase in superlinear fashion between around 100 and 100 000 services called by the same user.

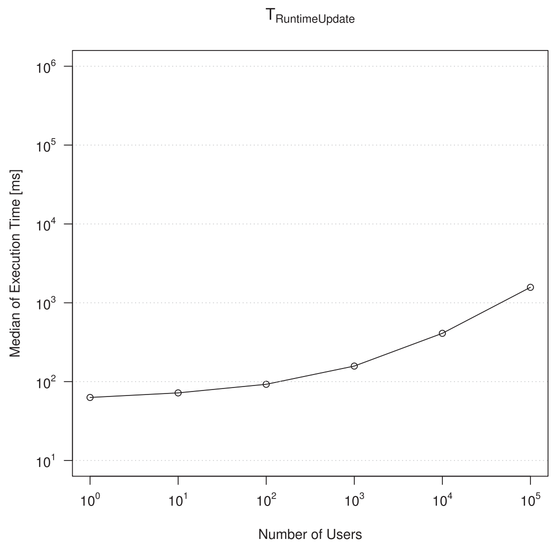
In order to interpret the results of the scalability experiments we conduct a worst case time complexity analysis of the transformations. In the following, we summarize the worst case time complexity analysis while further details are available in the iObserve GitHub repository. We introduce the following parameters.

Table 5
Clustering results with variance value 10.

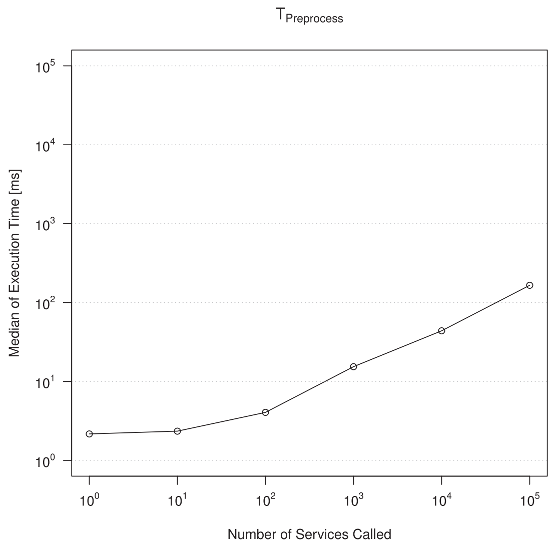
UG	UGM	Cl1	Cl2	Cl3	Cl4	MC	SSE
CR	50%	0	4000	0	0		
SKM	25%	2000	0	0	0	6.01%	141.32
SEM	25%	0	0	641	1359		
CR	25%	0	0	0	2000		
SKM	50%	886	3114	0	0	10.74%	134.33
SEM	25%	0	0	2000	0		
CR	25%	0	0	0	2000		
SKM	25%	0	0	2000	0	13.77%	159.15
SEM	50%	1215	2785	0	0		



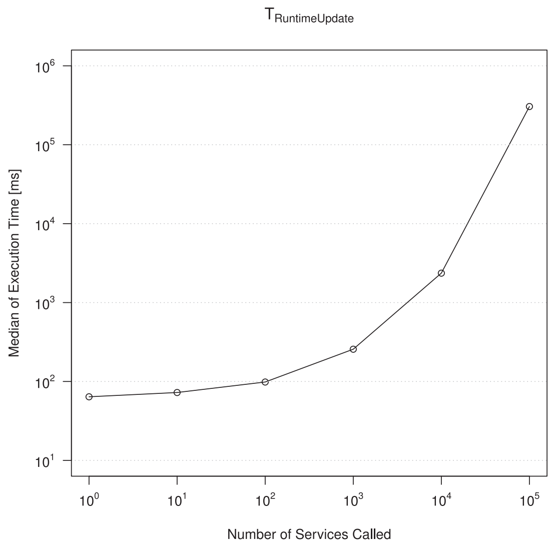
(a) Scalability results of $T_{Preprocess}$ for increased number of users calling the same service.



(b) Scalability results of $T_{RuntimeUpdate}$ for increased number of users calling the same service.



(c) Scalability results of $T_{Preprocess}$ for increased number of services called by the same user.



(d) Scalability results of $T_{RuntimeUpdate}$ for increased number of services called by the same user.

Fig. 10. Scalability results of the iObserve transformation pipeline.

- n: number of entry call events;
- s: number of user sessions;
- b: number of branch transitions;
- l: number of loops;
- e_s: number of elements in the largest user session;
- e_b: number of elements in the largest branch transition;
- e_l: number of elements in the largest loop;

First, the subtransformations of $T_{Preprocess}$ are explained. $T_{EntryCall}$ operates on hash maps to identify entry call events, which results in worst case time complexity of $O(1)$. $T_{EntryCallSequence}$ parses the class and operation signature of an entry call event and maps it to the corresponding user session based on its session information. As $T_{EntryCallSequence}$ operates on hash maps the worst case time complexity is $O(1)$. $T_{Preprocess}$ is executed for each observed event (in the experiments, for each entry call event) which results in worst case time complexity of $O(n)$.

$T_{RuntimeUpdate}$ is executed once for each experiment run. The subtransformations of $T_{RuntimeUpdate}$ are explained hereafter. $T_{UserGroupDetection}$ identifies the distinct calls within the user sessions, clusters the users sessions into user groups using the X-Means clustering algorithm (Pelleg and Moore, 2000) and calculates the usage intensity for each user group. This results in worst case time complexity of $O(s^2 * e_s)$ for $T_{UserGroupDetection}$.

For the users sessions of each user group $T_{BranchDetection}$ creates a tree structure by aggregating the contained entry calls, identifies the transition probability for each branch transition and then merges branch transitions and removes redundant branch transitions. This results in worst case time complexity of $O(s * log(s) + s * e_s * b^2)$. Note, we assume the number of user groups of the application will be limited to a small number. Therefore, we can neglect this factor in the worst case time complexity analysis.

$T_{LoopDetection}$ traverses the tree structure recursively for each branch transition to identify loops. This results in worst case time complexity of $O(b * e_b^3 + b * e_b * l)$.

$T_{ArchitecturalModelUpdate}$ iterates the tree structure to create the elements of the architectural model. This shows worst case time complexity of $O(b * (e_b + e_l))$.

For answering *RQ2.1*, we conclude as follows. $T_{Preprocess}$ scales well in a linear manner while increasing the number of users calling the same service. This is explained by aforementioned worst case time complexity analysis. In contrast, $T_{RuntimeUpdate}$ shows slightly superlinear scalability while increasing the number of users calling the same service. This can also be explained based on the worst case time complexity analysis as follows. In this specific experiment, the number of user sessions s increases the same way the number of entry calls n increases as each user session comprises exactly one call ($e_s = 1$). There is no loop ($l = 0, e_l = 0$) and no branch (which means number of branch transitions b is 1). This branch transition contains one element ($e_b = 1$). This results in quadratic scaling in the worst case depending on the number of user sessions s due to user group detection for this experiment.

For answering *RQ2.2*, we conclude as follows. $T_{Preprocess}$ scales well in a linear manner while increasing the number of services called by the same user as explained by the worst case time complexity analysis. In contrast, $T_{RuntimeUpdate}$ shows superlinear scalability while increasing number of services called by the same user. This is explained based on the worst case time complexity analysis as follows. Unlike the experiment for increasing the number of users calling the same service, which has to handle an increasing number of user sessions calling the same service, the experiment for increasing number of services called by the same user has to handle an increasing number of service calls (e_s) within a single user session ($s = 1$). There is no branch ($b = 1$). The CoCoME case study provides 13 different services. These 13 different services are called repeatedly in random order which results in many loops. This leads to cubic scaling in the worst case depending on the number of elements (e_b) to be processed for loop detection in this experiment.

$T_{RuntimeUpdate}$ shows long execution times for a large number of services called by the same user in the experiment. For smaller number of services called by the same user the experiment results in adequate execution times. We expect that more than some hundred service calls will very seldom occur in a single user session in a practical setting. Mostly, the number of services called will be far below. In conclusion, we expect $T_{RuntimeUpdate}$ will show adequate execution time for processing user sessions in practical settings.

5.5. *Threats to validity*

In case study research, four aspects of validity are distinguished (Runeson et al., 2012) – internal validity, external validity, construct validity, and conclusion validity (i.e., reliability).

Internal validity: A possible threat to validity is the set-based metrics JC and SRCC we applied for comparing models in the accuracy evaluation. In general, the order of elements in models may differ although they are interrelated the same way. For example, model elements referred by other elements within the model may be placed at different positions. Furthermore, deciding whether two models are equal is a non-trivial task in general. However, in the experiments on user behavior modeling we compare activity diagram like models with predecessor/successor relation between the model elements. We construct the reference models in a way that the order of the elements conforms to their predecessor and successor relationship (also for nested elements). This is why we can assume in the experiments that given two equal models, the order of elements within the models is equal. Also, in the experiments on changes in deployment and allocation we applied the set-based metric JC. In the allocation model and resource environment model of the PCM, there is no order between the execution containers and components deployed on an execution container. Consequently, we merely evaluate the existence of the elements in the experiments on changes in deployment and allocation for which the JC is an adequate metric.

External validity: In case study research, the representativeness of a sample case may be sacrificed to achieve a deeper understanding and better realism of the phenomena under study. Consequently, the results achieved in the study might not be transferable to an arbitrary other case, due to the individual properties of each case. However, the case study gives important insights and provides indicators for cases having similar properties. We used the CoCoME community case study as a study subject in our evaluation. The CoCoME case study balances real-world relevance with suitability for an academic environment and is established in the software architecture modeling and analysis community (Herold et al., 2008; Heinrich et al., 2015a). CoCoME comprises typical properties of a modern component-based software application running in the cloud (Heinrich et al., 2015a). Therefore, we expect a certain representativeness of the results achieved with CoCoME for other cloud-based software applications. Moreover, other case studies known in the community represent very similar web shop scenarios. Examples are Sock Shop[3] (emulates a web shop that sells socks), TeaStore (von Kistowski et al., 2018) (emulates a web shop for tea and tea supplies), JPetStore (Jung and Adolf, 2018b) (emulates a web shop that sells pets) and The Heat Clinic[4] (emulates a web shop for hot sauces). The reason for this is that e-commerce is probably the most relevant business case for cloud-based software applications.

Nevertheless, we can merely examine few sample cases in case study research. For example, the reference model for the clustering experiments represent per user group one of arbitrary combinations of service calls and transition probabilities. It is not feasible to examine each possible combination. Also in the scalability experiments we can only examine few of arbitrary combinations of service calls. Still the sample cases we have examined give indicators for the accuracy and scalability of iObserve. These can be further investigated in follow up experiments.

Construct validity: Evaluation results are highly dependent on the quality of the monitoring events. In the evaluation, we apply monitoring events generated from manually specified reference models instead of observing a running system. This is necessary to exclude influence factors of load drivers and the monitoring framework. Furthermore, generating the events allows for specifying any combination of events we want to investigate in the experiments.

Conclusion validity: While analyzing the evaluation results, the effects of interpretation by a specific researcher must be eliminated. In order to analyze the accuracy and scalability of iObserve, we apply statistical metrics which give a reasonable evidence and reduce the need for interpretation. Consequently, due to the study design, there is hardly an interpretation that may lead a researcher to another conclusion.

5.6. *Assumptions and limitations*

Currently, the iObserve approach 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 containers. These are the most common changes for cloud-based software applications discussed in literature (see discussion in Heinrich, 2016). iObserve can be

3 https://microservices-demo.github.io.

4 https://demo.broadleafcommerce.org.

extended for additional types of changes easily by adding new monitoring probe specifications, new transformations and new (partial) models or model elements to the iObserve 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 Platform as a Service (PAAS) cloud. Therefore, we assume we can observe all events needed from the perspective of an application developer and operator. Nevertheless, Software as a Service (SAAS)-based services can be represented in the architectural model and may be supported by additional monitoring technologies in the future development of iObserve.

A common limitation of runtime modeling approaches is the accuracy of the model depends on the length of the time span of observation. If the time span is too short, events that occur seldom may not be observed or probabilities calculated may be inaccurate due to neglected events.

6. Related work

Software adaptation and evolution is an architectural challenge (Kramer and Magee, 2007). Nevertheless, existing approaches dealing with the interplay between adaptation and evolution (e.g., Müller and Villegas, 2014; Oreizy et al., 2008) lack continuous modeling and updating of software architectures in component-based fashion.

There are various existing approaches related to iObserve which are discussed in this section. On the one hand, the novelty of iObserve is in the integration of several existing concepts to comprehensively support adaptation and evolution. On the other hand, iObserve is unique in the way that it updates development architectural models by observations of the running system and thus preserves development knowledge and, at the same time, reflects the current status of the running application. Table 6 gives and overview of the features of iObserve and how they are covered by existing approaches. If a feature is covered by a specific approach this is marked by "X" in the table. Details on the single approaches are discussed in the following.

Work related to the parts of iObserve presented in this paper can be distinguished basically into three major categories. First, work on reusing development models during operation. Second, work on model extraction from monitoring data. Third, work on user behavior modeling and user group detection.

Work on *reusing development models during operation*, such as Morin et al. (2009), Ivanovic et al. (2011), Schmieders and Metzger (2011) and Canfora et al. (2008), employs development models as foundation for reflecting software systems during operation. (Bencomo et al., 2014) gives an overview of runtime modeling and analysis approaches. The work in Morin et al. (2009) reuses sequence diagrams in order 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 modified. Consequently, operational changes like described in Section 2 are not supported. Other than this, the runtime models in Ivanovic et al. (2011), Schmieders and Metzger (2011) and Canfora et al. (2008) are modified during operation. These approaches employ workflow specifications created during development in order to carry out performance and reliability analyses during operation for service orchestration. 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 structure, which is required to reflect for example (de)replication of components (C3/C4).

Work on *model extraction* creates and updates model content during operation. Approaches, such as Schmerl et al. (2006), Song et al. (2011), van der Aalst et al. (2011), von Massow et al. (2011), Langhammer et al. (2016) and Walter et al. (2017), establish the semantic relation 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 from scratch by, e.g., observing and interpreting operation traces. Therefore, they disregard information that cannot be gathered from monitoring data, such as design perspectives on component structures and component boundaries. 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, the extracted states cannot be mapped to the initial application architecture. In consequence, the model hierarchy is flat and unstructured, which hinders software developers and operators in understanding the current situation of the application at hand. Furthermore, the work reflects processes but neither components nor their relations (C2 to C4). Other than this, the approaches in Schmerl et al. (2006) and (Song et al., 2011) extract components and their relations from events for the sake of comparing actual and intended architectures. To this end, the work modifies the runtime model by model transformation rules in response to single events. With these approaches we share the application of transformation rules to update a model based on monitoring events. The resulting models in Schmerl et al. (2006) and Song et al. (2011) are coarse-grained, which is sufficient for deciding whether an actual composition maps to the intended composition. However, when conducting analyses for identifying quality flaws (e.g., performance bottlenecks or confidentiality violations), the observation and reflection of resource consumption is crucial. Reflecting the consumption by the means of workload specifications requires to process event sets rather than single events, which outruns the event processing capacity of this approach (C1). The approaches in von Massow et al. (2011) and Brosig et al. (2011) extract component-based architectural models for performance simulation. von Massow et al. (2011) reflects component migration and (de)replication as well as execution container (de)allocation in the models. Brosig et al. (2011) extracts components, their connections and inner behavior as well as performance properties. However, also von Massow et al. (2011) and Brosig et al. (2011) neglect design perspectives on component structures and component boundaries. Furthermore, the extraction of workload specifications and user groups is not supported.

More sophisticated approaches for extracting architectural models can be found in Langhammer et al. (2016) and Walter et al. (2017). Langhammer et al. (2016) propose an approach to extract architectural, behavioral, and usage models from source code and test cases. In contrast to iObserve, the approach by Langhammer et al. is limited to static input for extracting the models as it does not use monitoring events. Therefore, this approach cannot reflect dynamic information like changes in deployment and usage during operation. Walter et al. (2017) proposes the PMX framework for extracting architectural performance models from monitoring events. In contrast to iObserve, PMX extracts architectural models from scratch which has aforementioned disadvantages for identifying components and their boundaries. Furthermore, an overview of existing work on software architecture reconstruction is given in Ducasse and Pollet (2009). In contrast to these approaches, iObserve does not aim to reconstruct a software architecture from source code but updates existing development models by monitoring data gathered during operation.

Table 6
Comparison of iObserve features to related approaches.

Features	iObserve	Reusing Dev. Model		Model Extraction					User Modeling		
		Morin et al. (2009)	Ivanovic et al. (2011), Schmieders and Metzger (2011) and Canfora et al. (2008)	Schmerl et al. (2006), Song et al. (2011) and Brosig et al. (2011)	von Massow et al. (2011)	van der Aalst et al. (2011)	Langhammer et al. (2016)	Walter et al. (2017)	Menascé et al. (1999) and Ruffo et al. (2004)	Vögele et al. (2018) and Jung and Adolf (2018a)	Kistowski et al. (2014) and Herbst et al. (2013)
Component-based architecture	X			X	X		X	X			
Preserve development knowledge	X	X	X				X				
Observe running system	X		X	X	X	X		X	X		
Reflect current status in model	X	X	X	X	X	X		X	X		
Identify upcoming quality flaws	X	X	X	X	X	X			X		
Extract user groups	X							X	X	X	
Extract workload specification	X							X		X	X
Detect migration	X				X						
Detect (de)replication	X				X						
Detect (de)allocation	X				X						

Work on *user behavior modeling and user group detection* extracts user behavior models and apply clustering algorithms to identify user groups from observed user interaction during operation. The architecture model extraction approach of Langhammer et al. (2016) also supports the extraction of user behavior specification in form of a PCM usage model. However, the approach in Langhammer et al. (2016) does not support the identification of different user groups. Furthermore, this approach does not support the extraction of usage intensities due to aforementioned limitation of static input for model extraction. Often Markov chains are applied for reflecting user interaction with the system (Li and Tian, 2003). For example, the approaches of Menascé et al. (1999) and Ruffo et al. (2004) apply Markov chains to extract user behavior models and rudimentary identify user groups based on log files. Based on our previous joint work (Vögele et al., 2015), the WESSBAS approach in Vögele et al. (2018) extracts workload specifications and approximates user groups from monitoring events for load testing and model-based performance prediction. The PMX framework (Walter et al., 2017) supports the extraction of different user groups and the approximation of workload specifications from monitoring events. In contrast to WESSBAS (Vögele et al., 2018) and PMX (Walter et al., 2017), the iObserve approach is based on a development model to be updated during operation. Therefore, in iObserve the user groups can be taken from the development model instead of approximating them. Moreover, WESSBAS is limited to workload specifications while PMX and iObserve also support the extraction, or update respectively, of architectural models. Jung and Adolf (2018a) proposes a first attempt for extracting behavior models which reflect real user groups instead of approximated groups. Since this approach also arose from the iObserve project it is similar to the approach proposed in this paper. However, it does not leverage knowledge about user groups contained in the development model.

Furthermore, there is work on modeling usage intensity without detecting user groups. LIMBO (Kistowski et al., 2014) allows for specifying usage intensity models for seasonal patterns, trends, bursts and noise. Herbst et al. (2013) propose an approach for forecasting usage intensities based on decision tree and direct feedback cycles. In contrast to these approaches, iObserve is focused on the identification of user groups and their behavior.

To summarize, development models that are reused during operation provide good comprehensibility to humans, but are not updated with respect to structural changes yet. However, structural updates are required to reflect migration (C2), (de)replication (C3/C4) and (de)allocation (C5/C6). Work on model extraction automatically creates runtime models from scratch, which may be useful for performance analysis for instance. However, as development decisions on the applications' architectures cannot be derived from monitoring events the resulting models lack architectural knowledge. Work on user behavior modeling and user group detection extract user groups and their behavior as well as usage intensities from monitoring events. These approaches create usage models from scratch as they lack leveraging knowledge contained in development models.

Work related to other parts of the iObserve approach is discussed in Heinrich (2016) and Heinrich et al. (2017).

7. Conclusion and future work

In this paper, we proposed models and transformations of the iObserve approach to aligning development-level evolution and operation-level adaptation of cloud-based software applications. We gave a detailed description of the RAC, a model to specify the correspondence between the elements of the architectural model and implementation artifacts. We came up with the iObserve transformation pipeline to update the architectural runtime model by changes observed during operation and proposed an approach to model complex workload based on these observations. In contrast to related approaches, we do not create runtime models from scratch but exploit and update architectural knowledge available from the application development. This avoids weak spots of common runtime modeling approaches like loss of architectural knowledge and different levels of abstraction of architectural models used in development and operation.

The accuracy of the architectural runtime models updated by the iObserve transformation pipeline has been evaluated for reflecting changes in workload characterization, component deployment and execution container allocation. Evaluation results show iObserve adequately represents monitoring events related to these changes during operation in architectural runtime models. Furthermore, we evaluate the scalability of the iObserve transformation pipeline. Evaluation results indicate the current implementation of iObserve adequately scales for some cases but shows scalability limits for other cases that must be further investigated in future work.

In the future, we plan to extend the scalability experiments by considering influence factors like variation in the complexity of user behavior in the experiments. Further revising and optimizing the current implementation of iObserve with respect to performance and scalability will also be part of future work.

Another topic of future work is to further expand the planning and execution phases of the MAPE control loop. First attempts have been proposed in Heinrich et al. (2017) and Pöppke (2017). Besides further analyzing design space exploration and optimization approaches to find optimal candidate architectural models, we will further investigate the execution of adaptation plans to allow for a maximum degree of automation where adaptation is possible without human intervention. We are currently developing an approach for observing and assuring geolocation compliance of data flows in cloud-based applications. This approach builds upon the modeling concepts proposed in this paper by making use of data-centric modeling and adaptation techniques. Another step in this direction is the investigation of data-centric models rather than control flow based models. In Seifermann et al. (2019), we proposed first attempts to extend architectural models by data properties and data flows which need to be revised and extended in the future.

Further, we plan to investigate the application of approaches from view-centric engineering with synchronized heterogeneous models, like (Kramer et al., 2015), in order to describe correspondence relations in a more fine-grained scale.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future — Managed Software Evolution (iObserve) and the German Federal Ministry of Education and Research under grant 01IS17106A (Trust 4.0). The author thanks all the members of the iObserve project Wilhelm Hasselbring, Reiner Jung, Andreas Metzer, Klaus Pohl, Ralf Reussner and Eric Schmieders for valuable discussions and their support. Special thanks to Nicolas Boltz for maintaining the iObserve tooling and supporting the evaluations described in the paper. The author also thanks Tobias Pöppke, David Peter and Alessandro Giusa who contributed to iObserve in the context of their bachelor and master theses.

References

Abramowitz, M., 1974. Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables. Dover.

Alsabti, K., 1998. An efficient k-means clustering algorithm. In: IPPS/SPDP Workshop on High Performance Data Mining.

Basili, V.R., Caldiera, G., Rombach, H.D., 1994. The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley.

Becker, S., Dencker, T., Happe, J., 2008. Model-driven generation of performance prototypes. In: Performance Evaluation: Metrics, Models and Benchmarks. Springer, pp. 79–98.

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

Brosig, F., Huber, N., Kounev, S., 2011. Automated extraction of architecture-level performance models of distributed component-based systems. In: 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 183–192.

Brun, Y., et al., 2009. Software Engineering for Self-Adaptive Systems. Springer, pp. 48–70, chapter Engineering Self-Adaptive Systems Through Feedback Loops.

Canfora, G., Di Penta, M., Esposito, R., Villani, M.L., 2008. A framework for QoS-aware binding and re-binding of composite web services. J. Syst. Softw. 81 (10), 1754–1769.

Cheng, B.H.C., et al., 2009. Software engineering for self-adaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems. Springer, pp. 1–26.

Ducasse, S., Pollet, D., 2009. Software architecture reconstruction: A process-oriented taxonomy. IEEE Trans. Softw. Eng. 35 (4), 573–591.

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

Hamlaoui, M.E., Trojahn, C., Ebersold, S., Coulette, B., 2014. Towards an ontology-based approach for heterogeneous model matching. In: 2nd International Workshop on the Globalization of Modeling Languages. CEUR, pp. 1–10.

Hasselbring, W., 2011. Reverse engineering of dependency graphs via dynamic analysis. In: 5th European Conference on Software Architecture: Companion Volume. ACM, pp. 5:1–5:2.

Hasselbring, W., Heinrich, R., Jung, R., Metzger, A., Pohl, K., Reussner, R., Schmieders, E., 2013. iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems. Technical Report 1309, Kiel University.

Heinrich, R., 2016. Architectural run-time models for performance and privacy analysis in dynamic cloud applications. SIGMETRICS Perform. Eval. Rev. 43 (4), 13–22.

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

Heinrich, R., Jung, R., Schmieders, E., Metzger, A., Hasselbring, W., Reussner, R., Pohl, K., 2015b. 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.

Heinrich, R., Jung, R., Zirkelbach, C., Hasselbring, W., Reussner, R., 2017. Software Architecture for Big Data and the Cloud. Elsevier, chapter An Architectural Model-Based Approach to Quality-aware DevOps in Cloud Applications.

Heinrich, R., Rostami, K., Reussner, R., 2016. The CoCoME Platform for Collaborative Empirical Research on Information System Evolution. Technical Report 2016,2; Karlsruhe Reports in Informatics, Karlsruhe Institute of Technology.

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

Herbst, N.R., Huber, N., Kounev, S., Amrehn, E., 2013. Self-adaptive workload classification and forecasting for proactive resource provisioning. In: 4th ACM/SPEC International Conference on Performance Engineering. ACM, pp. 187–198.

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

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

Jung, R., Adolf, M., 2018a. Extracting realistic user behavior models. In: CEUR Vol-2066.

Jung, R., Adolf, M., 2018b. The JPetStore suite: A concise experiment setup for research. In: 10th Symposium on Software Performance. pp. 1–3.

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

Kendall, M.G., 1938. A new measure of rank correlation. Biometrika 30 (1/2), 81–93.

Khan, A., Yan, X., Tao, S., Anerousis, N., 2012. Workload characterization and prediction in the cloud: A multiple time series approach. In: Network Operations and Management Symposium. IEEE, pp. 1287–1294.

Kistowski, J.v., Herbst, N.R., Kounev, S., 2014. Modeling variations in load intensity over time. In: 3rd International Workshop on Large Scale Testing. ACM, pp. 1–4.

Koziolek, A., Koziolek, H., Reussner, R., 2011. Peropteryx: Automated application of tactics in multi-objective software architecture optimization. In: ACM SIGSOFT Conference on Quality of Software Architectures. ACM, pp. 33–42.

Kramer, M.E., Langhammer, M., Messinger, D., Seifermann, S., Burger, E., 2015. Change-driven consistency for component code, architectural models, and contracts. In: 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering. ACM, pp. 21–26.

Kramer, J., Magee, J., 2007. Self-managed systems: an architectural challenge. In: Future of Software Engineering. pp. 259–268.

Kunz, J., Heger, C., Heinrich, R., 2017. A generic platform for transforming monitoring data into performance models. In: 8th ACM/SPEC on International Conference on Performance Engineering. pp. 151–156.

Langhammer, M., Shahbazian, A., Medvidovic, N., Reussner, R.H., 2016. Automated extraction of rich software models from limited system information. In: 13th Working IEEE/IFIP Conference on Software Architecture. pp. 99–108.

Levandowsky, M., Winter, D., 1971. Distance between sets. Nature 234 (5).

Li, Z., Tian, J., 2003. Testing the suitability of Markov chains as web usage models. In: 27th Annual International Computer Software and Applications Conference. pp. 356–361.

Lientz, B.P., Swanson, B.E., 1980. Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley, pp. 1–160.

Menascé, D.A., Almeida, V.A.F., Fonseca, R., Mendes, M.A., 1999. A methodology for workload characterization of e-commerce sites. In: 1st ACM Conference on Electronic Commerce. ACM, pp. 119–128.

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

Mueller, E., 2019. What is devops? http://theagileadmin.com/what-is-devops/.

Müller, H., Villegas, N., 2014. Runtime evolution of highly dynamic software. In: Evolving Software Systems. Springer, pp. 229–264.

Murphy, G., Notkin, D., Sullivan, K., 2001. Software reflexion models: bridging the gap between design and implementation. IEEE Trans. Softw. Eng. 27 (4), 364–380.

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

Pelleg, D., Moore, A.W., 2000. X-means: Extending k-means with efficient estimation of the number of clusters. In: 17th International Conference on Machine Learning. pp. 727–734.

Peter, D., 2016. Observing and Modeling Workload Characteristics of Dynamic Cloud Applications (Master's Thesis). Karlsruhe Institute of Technology.

Pöppke, T., 2017. Design Space Exploration for Adaptation Planning in Cloud-based Applications (Master's Thesis). Karlsruhe Institute of Technology.

Qin, C., Eichelberger, H., 2016. Impact-minimizing runtime switching of distributed stream processing algorithms. In: EDBT/ICDT Workshops 2016.

Reussner, R.H., et al. (Eds.), 2016. Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press, ISBN: 978-0-262-03476-0.

Ruffo, G., Schifanella, R., Sereno, M., Politi, R., 2004. WALTy: a user behavior tailored tool for evaluating web application performance. In: 3rd IEEE International Symposium on Network Computing and Applications. pp. 77–86.

Runeson, P., Host, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. Wiley.

Sadou, S., Allier, S., Sahraoui, H., Fleurquin, R., 2011. From object-oriented applications to component-oriented applications via component-oriented architecture. In: Working IEEE/IFIP Conference on Software Architecture. pp. 214–223.

Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H., 2006. Discovering architectures from running systems. IEEE Trans. Softw. Eng. 32 (7), 454–466.

Schmieders, E., Metzger, A., 2011. Preventing performance violations of service compositions using assumption-based run-time verification. In: Towards a Service-Based Internet. Springer, pp. 194–205.

Schmieders, E., Metzger, A., Pohl, K., 2015. Runtime model-based privacy checks of big data cloud services. In: Service-Oriented Computing. Springer, pp. 71–86.

Seifermann, S., Heinrich, R., Reussner, R., 2019. Data-driven software architecture for analyzing confidentiality. In: IEEE International Conference on Software Architecture. IEEE, pp. 1–10.

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

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

van der Aalst, W., Schonenberg, M., Song, M., 2011. Time prediction based on process mining. Inf. Syst. 36 (2), 450–475.

van Hoorn, A., Rohr, M., Hasselbring, W., 2008. Generating probabilistic and intensity-varying workload for web-based software systems. In: SPEC Int'l Performance Evaluation Workshop. In: LNCS, Springer, pp. 124–143.

van Hoorn, A., Vögele, C., Schulz, E., Hasse bring, W., Krcmar, H., 2014. Automatic extraction of probabilistic workload specifications for load testing session-based application systems. In: 8th International Conference on Performance Evaluation Methodologies and Tools. ACM, pp. 139–146.

Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H., 2013. Typing artifacts in megamodeling. Softw. Syst. Model. 12 (1), 105–119.

Vogel, T., Giese, H., 2010. Adaptation and abstract runtime models. In: Workshop on Software Engineering for Adaptive and Self-Managing Systems. ACM, pp. 39–48.

Vogel, T., Giese, H., 2014. On unifying development models and runtime models (position paper). In: 9th Int'l Workshop on Models at Run.Time. CEUR.

Vögele, C., Heinrich, R., Heilein, R., Krcmar, H., van Hoorn, A., 2015. Modeling complex user behavior with the palladio component model. In: Softwaretechnik-Trends, Vol. 35(3).

Vögele, C., van Hoorn, A., Schulz, E., Hasse bring, W., Krcmar, H., 2018. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. Softw. Syst. Model. 17 (2), 443–477.

von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S., 2018. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In: 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems. IEEE, pp. 223–236.

von Massow, R., van Hoorn, A., Hasselbring, W., 2011. Performance simulation of runtime reconfigurable component-based software architectures. In: ECSA. In: LNCS, vol. 6903, Springer, pp. 43–58.

Walter, J., Stier, C., Koziolek, H., Kounev, S., 2017. An expandable extraction framework for architectural performance models. In: 3rd International Workshop on Quality-Aware DevOps. ACM.

Wayne, D., 1990. Spearman rank correlation coefficient. In: Applied Nonparametric Statistics, second ed..

Wulf, C., Wiechmann, C.C., Hasselbring, W., 2016. Increasing the throughput of pipe-and-filter architectures by integrating the task farm parallelization pattern. In: 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering. IEEE, pp. 13–22.

Yie, A., Casallas, R., Deridder, D., Wagelaar, D., 2012. Realizing model transformation chain interoperability. Softw. Syst. Model. 11 (1), 55–75.

Robert Heinrich is head of the Quality-driven System Evolution research group at Karlsruhe Institute of Technology (Germany). He holds a doctoral degree from Heidelberg University and a degree in Computer Science from University of Applied Sciences Kaiserslautern. His research interests include modularization and composition of model-based analysis for several quality properties, such as performance, confidentiality and maintainability, as well as for several domains, such as information systems, business processes and automated production systems. One core asset of his work is the Palladio software architecture simulator. He is involved in the organization committees of several international conferences, established and organized various workshops, is reviewer for international premium journals, like IEEE Transactions on Software Engineering and IEEE Software, and is reviewer for international academic funding agencies. Robert is principal investigator or chief coordinator in several grants from the German Research Foundation and governmental funding agencies. He has (co-) authored more than 50 peer-reviewed publications and spent research visits in Chongqing (China) and Tel Aviv (Israel).