INSTITUT FÜR INFORMATIK

iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems

Wilhelm Hasselbring, Robert Heinrich,
Reiner Jung, Andreas Metzger, Klaus Pohl,
Ralf Reussner, Eric Schmieders

Bericht Nr. 1309

October 2013

ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems

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

[1]Software Engineering Group, Kiel University
[2]Software Design and Quality, Karlsruhe Institut of Technology
[3]Software Systems Engineering, University of Duisburg-Essen

October 2013

Contents

1 Project Objectives

The goal of iObserve is to develop methods and tools to support evolution and adaptation of long-living software systems. Future long-living software systems will be engineered using third-party software services and infrastructures. Key challenges for such systems will be caused by dynamic changes of deployment options on cloud platforms. Third-party services and infrastructures are neither owned nor controlled by the users and developers of service-based systems. System users and developers are thus only able to observe third-party services and infrastructures via their interface, but are not able to look into the software and infrastructure that provides those services. This in turn leads to significant research challenges with respect to the observation and analysis of those systems behavior to detect anomalies.

2 Data Migration Scenario

The development and availability of cloud services fostered the transition from privately used dedicated servers to settings with dynamically allocated private and public cloud services to improve resource efficiency. This transition affects compositional properties of the software systems architecture, will impact performance predictions, and introduces data security and privacy constraints on the data processed by the software. In industry, healthcare, or governmental institutions, for instance, the protection of data and the conformance to data policies is of high relevance.

A major reason for resistance to use third-party software services and infrastructures is insufficient trust in these services and infrastructures. The services and infrastructures need to be trustworthy to its users [8]. Software trustworthiness consists of several attributes, such as reliability, availability, performance and security [1]. For our example adaptation scenario, we consider the properties cost, performance, and privacy as well as their mutual influences.

The elasticity of cloud infrastructures is achieved by the exploitation of adaptation capabilities among the involved software components. A database service, that provides certain adaptation capabilities such as replication, may be migrated to some low-cost data center, located in another country via moving its entire deployment context (the VM-instance). This migration may lead to accidental disclosures of confidential data, which violates laws such as the EU data protection directive[1]. Consequently, a challenge is to systematically share knowledge across multiple software components. This is relevant for carrying out adaptation actions compliant to privacy and also cost constraints.

According to some legislation, such as the "Bundesdatenschutzgesetz"[2] in Germany, person-related data must only be stored and processed in European countries that assert certain data privacy regulations. Figure 2.1 illustrates a scenario in which data-location policies are violated: The supermarket system of the CoCoME (see [11]) case study is, initially, deployed on a German cloud storage provider; then – in the face of high workload – the German provider allocates additional resources from an Italian provider. This Italian provider in turn allocates low-cost resource from a US provider to replicate the data of the German provider for increased performance. As a consequence, some German data policy is violated. For the German provider, it is intricate to identify such violations, which can be considered as data-flow anomalies. Based on this example scenario, iObserve investigates new integrated observation methods and techniques.

Once migrated to another data center, the receiving third-party has to gain knowl-

[1] http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML
[2] http://www.gesetze-im-internet.de/englisch_bdsg/federal_data_protection_act.pdf

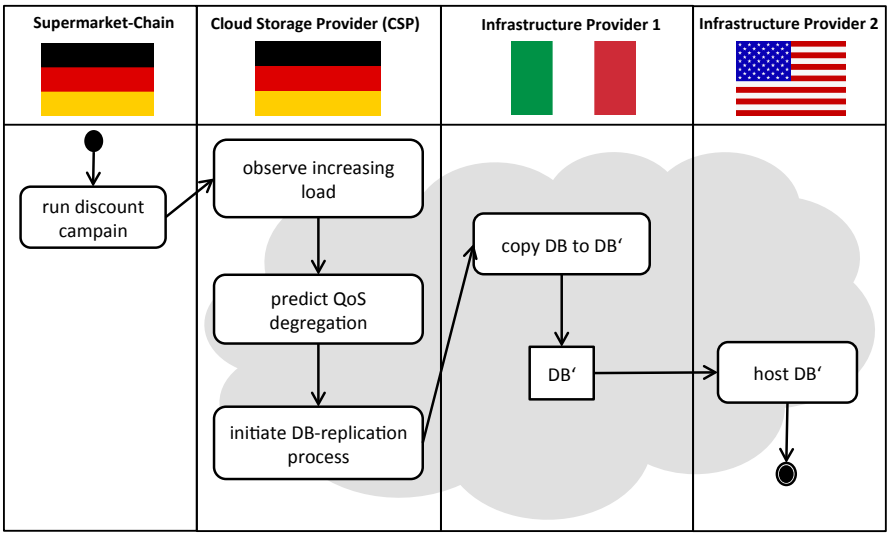

Figure 2.1: iObserve adaptation scenario example for observation of data-migration processes in the CoCoME case study

edge about the performance and privacy goals associated to the received data in order to appropriately cope with anomalies during runtime. Anomalies, such as performance degradations or unusual data flows, encountered at runtime should be handled to avoid penalties defined in service level agreements (SLA). Consequently, another challenge is, for instance, the selection of appropriate mitigation actions, which may require autonomic adaptation routines or the evolution of the software application.

3 Overall Goals

In iObserve, we address the challenges of relying on third-party services. To gain knowledge on the performance and data constraints of an application executed in the cloud, we develop innovative methods and techniques to monitor third-party and in-house services utilizing cloud infrastructure. Based on the analysis of the monitoring data, we populate models@runtime, which are subsequently used to detect anomalous behavior, such as data policy violations, performance degradation or risk of cost increases. Specifically, we study the following issues:

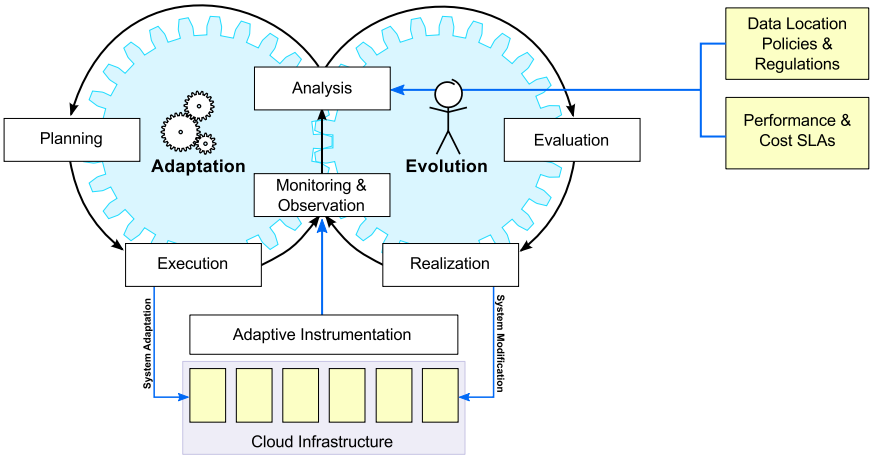

Figure 3.1: iObserve adaptation and evolution life-cycle, in which evolution and adaptation are two mutual, interwoven cycles that influence each other

- New aspect-oriented, model-driven monitoring methods and techniques [14, 17] for applications and services in the cloud to observe performance, cost and privacy properties will be designed and evaluated. For cloud-based systems, the management of deployment options spans a wide range of challenges in software and service engineering [7]. In particular, the selection of appropriate cloud deployment options constitutes a multi-objective optimization problem [5]. Continuous monitoring of dynamic cloud-based systems allows for appropriate adaptation and evolution of cloud deployments.
- New modeling methods and techniques for application and service behavior based on monitoring data to fuel the analysis with prediction information gained from

model simulation and analysis. The approach of models@runtime is relevant in this context, since observation data may be fed into such models@runtime.

- New analysis methods and techniques for the monitoring data to detect anomalies [4, 9] in various properties of cloud-based applications.

Figure 3.1 illustrates the system life-cycle, in which evolution and adaptation are two mutual, interwoven cycles that influence each other. Evolution activities are performed by software engineers, while adaption activities are usually pre-planned and automatically performed by some control software components. For our example for the CoCoME case study in Section 2, the data location policies and the performance SLAs drive the analysis procedures.

The iObserve project addresses those challenges by following a model-based approach. Consequently, the overall goal of the project is to develop and validate new models and techniques for runtime observation and anomaly detection of future service-based software systems deployed on third-party platform and infrastructure services, through extending and integrating previous work on monitoring, benchmarking, and meta-modeling. Achieving this goal will be crucial for future large-scale software systems to empower their continuous modification over time – a prerequisite for a long life-time.

4 Kieker Palladio Integration

In software development and maintenance, adaptation and evolution are the two key activities to allow software to provide the necessary functionality in the right quality. In iObserve, one focus is to observe software at runtime and fuel the decision to adapt or evolve the software system. The evaluation of software is based on observations and forecasting models. These models can be constructed at runtime with architecture recovery mechanism. However, these models do only represent those parts of a system which have been used. A solution to the partial models are models of the design phase which include all components. Palladio [3] provides such design phase models. It provides model elements and simulation behavior to forecast quality attributes such as performance and maintainability. The parameters for these forecasting rules can be adjusted by monitoring which is the domain of Kieker [17, 16, 12].

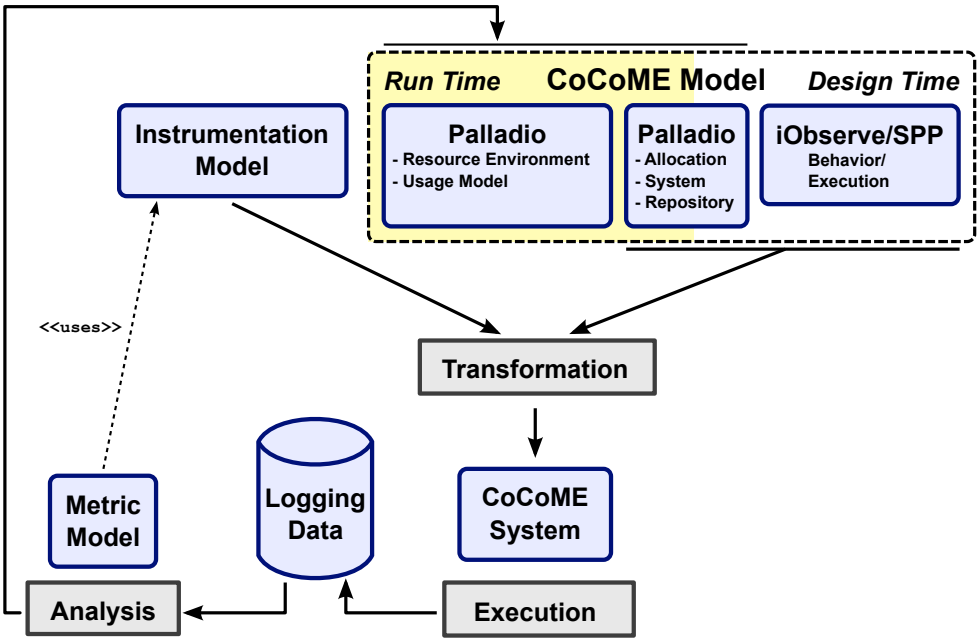

Figure 4.1: Kieker Palladio integration overview

This constellation and promising results from the SLAstic approach [15, 13] encourage us to integrate Kieker and Palladio forming a joint platform for software design

and evolution. An overview of the integration is given in Figure 4.1. The CoCoME model is key in this figure. It specifies software systems, deployment, resources, and usages from an architectural and behavioral perspective. In our work we focus on four objectives. First, introducing monitoring probes in software systems must be described on a model level to be able to be combined with Palladio models. Second, the modeled instrumentation must be introduced in the system code. Third, the evaluation of the monitoring results based on metrics must be integrated with Kieker. And fourth, the analysis results must be fed back into the prediction and forecasting part of Palladio models in order to reflect changed performance configurations.

This requires complete meta-modeling of the software system including quality aspects such as performance, geo-location, and costs. Business objectives, such as data-migration-policies, must be associated to monitoring probes. Therefore, we are constructing a metrics-meta-model describing the relationship between base measures and derived measures. Existing meta-models related to software metrics[1] may be a starting point for this. Continuous meta-modeling enables traceability among the hardware level, the software architecture level, and the business level. Traceability comprises the relationship between a derived measure and its components (i.e., base measures and/or derived measures) within the measure meta-model. Moreover, traceability comprises the relation between a base measure in the metric-meta-model, and the related value of an attribute in the system model, as well as between a derived measure in the metrics-meta-model, and the related analysis result.

4.1 Model-driven Instrumentation and Monitoring

Instrumentation of software systems for monitoring purposes is a complex technology dependent task. To integrate different technologies and technological solutions, a common approach must be found. Therefore, we divided instrumentation in five distinct layers addressing parts of the instrumentation problem.

Figure 4.2 shows these five layers. The top most layer describes the data structures for the monitoring, which resemble a rather flat, record like structure realizable in any underlying programming language.

The second layer addresses the placement of instrumentation probes in relation to software components. Depending on the realization technology of an application, e.g., J2EE or Spring, different configuration and code must be used to realize this placement. The Kieker framework provides probes for several technologies, but in their present implementation are not record agnostic.

The lower two layers in Figure 4.2 are provided by the Kieker framework, which supports logging and serialization for JVM and Perl-based software and provides the means to realize both with other languages.

As a common notation for the monitoring data, we developed an instrumentation record language (IRL) realized with Xtext[2]. Listing 4.1 provides an example of the declaration of monitoring records in IRL. The language comes with a generic set of

[1] http://www.omg.org/spec/SMM/
[2] http://www.eclipse.org/Xtext/

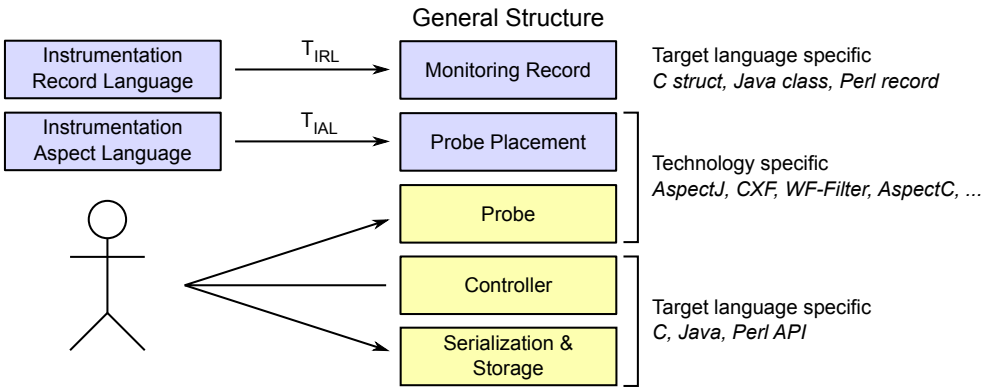

Figure 4.2: Models, artifacts and runtime components of model-driven instrumentation

primitive types and a definition of a type-mapping for different programming languages. Furthermore, it allows to include data types from foreign type systems.

```
package kieker.records

use ecore "http://www.eclipse.org/emf/2002/Ecore"

struct AbstractOperationEvent {
    const NO_SIGNATURE = "<no signature>"
    long timestamp = -1
    long traceId = -1
    int orderIndex = -1
    string operationSignature = NO_SIGNATURE
    string classSignature = NO_SIGNATURE
}

struct BeforeOperationEvent extends AbstractOperationEvent {}
struct AfterOperationEvent extends AbstractOperationEvent {}
struct CallOperationEvent extends AbstractOperationEvent {
    string calleeOperationSignature = NO_SIGNATURE
    string calleeClassSiganture = NO_SIGNATURE
}
struct AfterOperationFailedEvent extends AfterOperationEvent {
    string exception = "<invalid exception>"
}
```

Listing 4.1: Kieker Instrumentation Record Lanugage

The IRL also provides sub-typing of record structures and allows to define default values for record properties.

The second language, addresses the application of probes to a software model and is called instrumentation aspect language (IAL). Listing 4.2 contains an sample of the current IAL describing the application of before and after event probes to all methods of the `TradingSystem` model of CoCoME.

```
package org.spp.cocome.instrumentation

probe /TradingSystem/**/* *(*) : * {
    before collect BeforeOperationEvent(time, id, index, ./name, ../name)
    after collect AfterOperationEvent(time, id, index, ./name, ../name)
}
```

Listing 4.2: Kieker Instrumentation Aspect Lanugage

The IAL uses an XPath[3] and AspectJ[4] like syntax to describe queries over the model of a software system. It supports the usual three aspect weaving types before, after and around. For each weaving point, a monitoring record type can be specified. For its initialization, runtime sources, like `time` or `id`, can be used as well as model properties.

4.2 Code Generation

Code generation is a cornerstone to provide our CoCoME-based case study. The two main properties of this endeavor are multiple crosscutting concerns and continuous modifications to models and meta-models evolved in the code generation. This results in complex code generation which has to be modified repeatedly to serve our case study development. Especially related projects in the SPP 1593, which address model-code co-evolution, require an adaptable code generator. As a solution we are developing a process allowing to develop code generators for each aspect separately where the combination of code generation is performed by an automated process resulting in lesser complex partial generators in the developer's scope.

In general there are three different code generation scenarios depending on source and target meta-models, as shown in Figure 4.3. The vertical arrows in the figure represent transformations, while the horizontal arrows represent the direction of references between models. The first scenario illustrates a situation where the base and the aspect code get transformed separately into separate target models and an additional weaver combines them based on reference information. The second scenario is quite similar, but the reference direction is inverted. Meaning, the information about the weaving has moved from the aspect to the base model. And the third, scenario describes direct weaving of base and aspect models. For our case study based on CoCoME, we rely on the first scenario, which fits perfectly for a Java and AspectJ environments.

The main obstacle of weaving target base and target aspect models is to determine the corresponding weaving locations (end of the reference) in the target model. The location could be determined by inspecting the code of the base model code generator, but every time it changes, the location resolver in the aspect code generator must be adapted. A situation we want to avoid. Therefore, the location must be determined

[3] http://www.w3.org/TR/xpath/
[4] http://eclipse.org/aspectj/

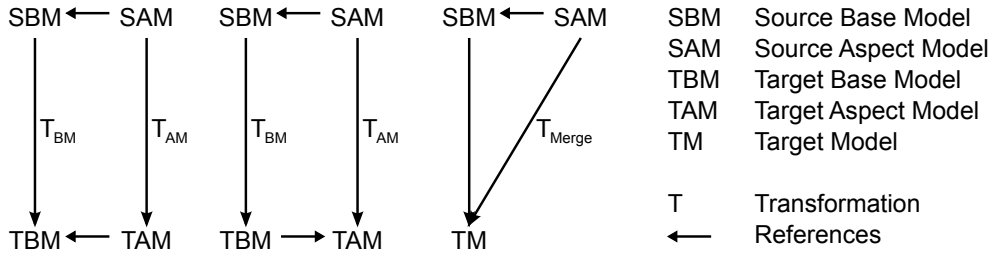

Figure 4.3: Three different aspect and base model code generation scenarios

differently.

As a starting point, we know the reference end in the source model, as it can be inferred by using the aspect selection query and determine source base model nodes. To find the corresponding target model nodes, we must trace all target model nodes created during transformation, as illustrated in Figure 4.4.

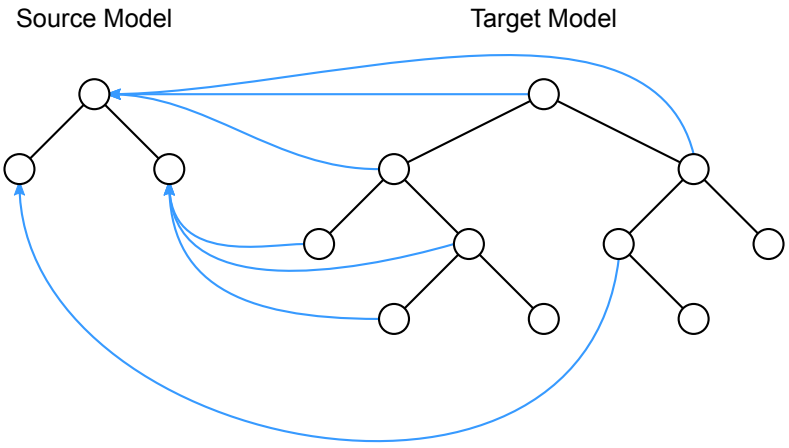

Figure 4.4: Traceability of target to source model nodes

However, there could be more target model nodes for one source model node. Therefore, we require a selection criteria to choose from the related nodes. These selectors are formulated as model queries and solely based on the target meta-model. Therefore, they can be reused for other generators and different versions of generators. Right now these selectors must be implemented in Xtend, but we will develop a specialized notation.

4.3 Evaluation of Monitoring Results

In Section 4.1, we introduced instrumentation languages to describe monitoring data and the instrumentation aspect itself. As illustrated in Figure 4.1, the monitoring results in monitoring logs, which must be interpreted to determine what and how to change the Palladio performance models.

```
model pcm "http://sdq.ipd.uka.de/PalladioComponentModel/5.0"

collect AverageMethodResponseTime ( String methodName )
   average AfterOperationEvent − BeforeOperationEvent
   scope ( pcm.repository.Operationsignature.entityName == methodName )

measure BeforeOperationEvent
measure AfterOperationEvent
```

Listing 4.3: Evaluation metrics based on MAMBA [6]

Our solution for these interpretations, rely on the measure definition language (MDL) from our MAMBA approach [6]. Listing 4.3 shows a simple average response time metric for operation signatures based on the Palladio meta-model. It uses measures defined in the instrumentation aspect language.

Right now, the cooperation between the instrumentation and measure languages is solely based on names, which are not checked by the editors. However, our next steps include the integration of these languages providing a closed tool-chain from instrumentation to evaluation.

5 Enforcing Data Geo-Location Policies amongst multiple Stakeholders

In the scenario above, privacy data collected by the discounter must not be transferred to the US as the transfer would violate German laws. To support CSPs in jointly fulfilling geo-location specifications iObserve elaborates an approach which is able to support cloud stakeholders in taking decisions on data transfer requests automatically. Under the application of runtime verification our approach constraints planned migrations such that privacy data is prevented from being transferred to locations excluded by related data-location policies.

In order to elaborate the constraint more precisely, which our work imposes on data transfer, let us assume that a set of components C (e.g. databases or analytic services) is deployed on data centers, which have certain geo-locations L, expressed with $HasGeoLocation(C, L)$. Components have access to each other, which can be modeled as directed graph with a set of ordered component pairs, $p = (c_a, c_b) \in C \times C$, i.e. $Access(c_a, c_b)$. Without having further information on the access we assume the 'worst case', which means that an access is transitive, i.e. $Access(c_a, c_b) \wedge Access(c_b, c_c) \rightarrow Access(c_a, c_c)$. Furthermore, the privacy data called personally identifiable information (pii) I is being processed (e.g. business intelligence operations or persistency) in components, expressed with $Processed(I, C)$. pii is under the governance of geo-location policies, which postulate that I shall not be processed in the excluded geo-locations.

Based on work, such as [10], our approach equips an independent Trust Authority with a Policy Decision Point (PDP) component. The CSPs will inform the PDP about planned actions. Kieker is used to complement the information required for taking the decision (such as location data), whereas Palladio will be used to build the model@runtime. During runtime the PDP checks the runtime model against the geo-location constraint

$$\begin{aligned} \forall c_a \forall c_b \forall g_a \forall g_b HasGeoLocation(c_a, g_a) \wedge HasGeoLocation(c_b, g_b) \wedge Access(c_a, c_b) \\ \wedge Processed(I, c_b) \models \neg \exists (g_a, g_b \in E) \end{aligned} \tag{5.1}$$

with $c_a, c_b \in C$ and $g_a, g_b \in L$, such that no component c shall process any pii I at an excluded location $g \in E$. By this the PDP accepts or declines the announced action such that no privacy data is going to be processed at excluded locations.

The challenges of this work are twofold. First we have to develop or adopt an appropriate decision approach, which finds a balance between accepting and declining actions. Accepting actions potentially lead to law violations whereas declining actions may leave cloud benefits such as reduction of idle times unused. Second we have to find means to verify the runtime model against the data-geo-location polices, i.e. to execute the constraint above. To this aim we explore established techniques such as model checking or constraint solving for their applicability in this setting.

We are currently elaborating the overall approach which connects Kieker, Palladio and the PDP-component to reflect the scenario sketched above. After defining their interfaces we now work on the decision approach being responsible for deciding on planned actions such that privacy data remains within the specified geographical boundaries.

6 Comparision of CoCoME Design and Realizations

The Common Component Modeling Example (CoCoME) [11] was developed as a common model to evaluate and compare modeling methods. It was chosen as one of the common case studies for the SPP 1593[1]. As part of the SPP 1593 we are also committed to use the CoCoME case study as the basis of our evaluation.

There are different implementations and models of CoCoME available. As we need a complete and correct realization of CoCoME, we selected a Palladio model and two implementations of CoCoME. We then compared these artifacts with the CoCoME design documentation and found some discrepancies. In this section, we explain our comparison approach, document our findings, and describe our next step to come to a complete CoCoME model and implementation.

Reverse Engineering CoCoME: The primary goal of the CoCoME case study is to provide a common frame for all SPP 1593 projects which utilize an enterprise application scenario. The CoCoME implementation[2] is based on Java and uses RMI[3] and ActiveMQ[4] for communication. As most of the project in the SPP require a model of CoCoME which is in sync with the code, our first step is to determine the structure and behavior of the CoCoME implementation with a combined static and dynamic analysis.

Due to the nature of RMI we were not able to create traces across services. However, internal traces could be determined, allowing us to analyze the behavior and generate component diagrams. We complied out of this information the structural view of the CoCoME implementation (cmp. Figure 6.1).

Furthermore, we analyzed the conformance of the CoCoME code to coding conventions and compliance to the architecture in the CoCoME design [11]. The compliance to the architecture was high on a component level, but not on the service composition. Furthermore, Java coding conventions were not met and the implemented protocols deviated from the design. However, the code can be refactored to meet coding conventions and reused for the SOA-fication of the project.

Analyzing the Palladio Model of CoCoME: The modeling notation for the CoCoME case study is the Palladio Component Model (PCM) [2]. In recent years, the 'Software

[1] http://www.dfg-spp1593.de/
[2] http://sourceforge.net/apps/trac/cocome/
[3] http://jcp.org/en/jsr/detail?id=78
[4] http://activemq.apache.org/

Design and Quality' chair realized a CoCoME model in Palladio. Therefore, we first investigated this solution, before starting our own modeling process.

The Palladio model of CoCoME focuses on the services of CoCoME, modeling them in great detail. However, cash desk lines where not modeled. And the complete behavior of CoCoME is not modeled in the Palladio model.

Comparing Results: The existing code and model of CoCoME represent different parts of the CoCoME design, as shown in Figure 6.1.

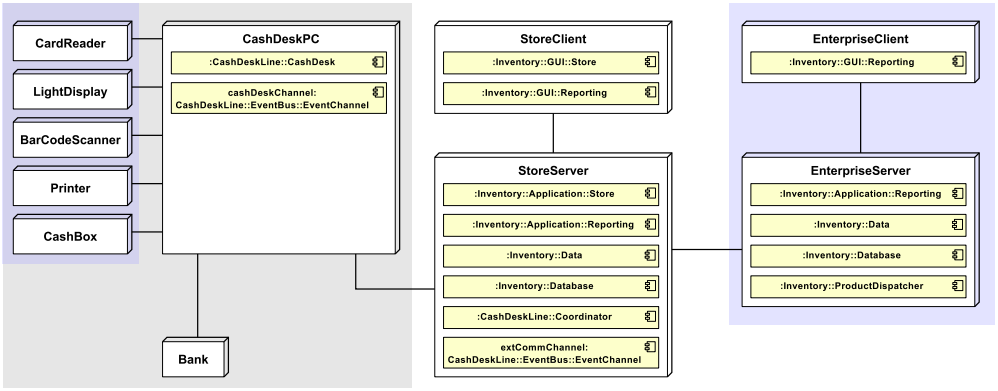

Figure 6.1: CoCoME deployment diagram and the coverage of Palladio and the implementation. gray = not covered by Palladio, blue = not covered by the implementation

Both realizations, the Palladio model and the implementation, do not cover the complete CoCoME design. Therefore, no complete model of CoCoME is available at the moment. However, information from our reverse engineering can be used to fill the gap in the Palladio model in conjunction with the CoCoME design document.

7 Extending Palladio by a Data and Expression Languages

The PCM is a powerful tool for describing software architectures. Among others, it provides model elements to specify software components, their deployment, and the communication between them. Once code is generated, it can be executed and monitored. However, Palladio lacks a formal execution semantic to describe operations and data models. Execution semantics of operations and data models are required in order to generate executable code.

In order to address these shortcomings we are going to extent the expressiveness of PCM in terms of an expression language and semantics required for code generation.

Extending Palladio with Expression Languages: Our central goal is to develop an approach which establishes permanent consistency between model and code. As the PCM is not able to describe functionality in an executable manner and the data modeling is not part of the meta-model, we will augment Palladio accordingly. Therefore, we develop an entity language in conjunction with a query language to address data-modeling. Furthermore, we supplement the PCM with a DSL to model internal data structures and method bodies.

```
package org.spp.cocome

entity Product {
        int id
        string name
        currency price
}

entity StockItem {
        Product product
        int stock
}

entity Cart {
        CartItem[] item
}

CartItem {
        Product product
        int stock
}
```

Listing 7.1: Extention Language Example

We evaluate our approach based on the CoCoME example. For the aim of evaluation

we complement the Palladio model of CoCoME. In a first step, we specified several entities relevant in CoCoME. The entity language is illustrated in Listing 7.1.

Semantics for Palladio The code generation for CoCoME will be realized by a code generator based on the Palladio meta-model and the DSLs. As target environment we use J2EE[1] and its technologies including JPA[2] and JSF[3].

[1] http://jcp.org/en/jsr/detail?id=316
[2] http://jcp.org/en/jsr/detail?id=317
[3] http://www.jcp.org/en/jsr/detail?id=344

Bibliography

[1] Steffen Becker, Marko Boskovic, Abhishek Dhama, Simon Giesecke, Jens Happe, Wilhelm Hasselbring, Heiko Koziolek, Henrik Lipskoch, Roland Meyer, Margarete Muhle, Alexandra Paul, Jan Ploski, Matthias Rohr, Mani Swaminathan, Timo Warns, and Daniel Winteler. Trustworthy software systems: A discussion of basic concepts and terminology. ACM SIGSOFT Software Engineering Notes, 31(6):1–18, 2006.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In WOSP '07: Proceedings of the 6th international workshop on Software and performance, pages 54–65, New York, NY, USA, 2007. ACM.

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. Journal of Systems and Software, 82:3–22, 2009.

[4] Jens Ehlers, André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In Proceedings of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC 2011), pages 197–200. ACM, June 2011.

[5] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In 35th International Conference on Software Engineering (ICSE 2013), pages 512–521. IEEE Press, May 2013.

[6] Sören Frey, André van Hoorn, Reiner Jung, Benjamin Kiel, and Wilhelm Hasselbring. MAMBA: Model-based analysis utilizing OMG's SMM. In Proceedings of the 14. Workshop Software-Reengineering (WSR '12), May 2012.

[7] John Grundy, Gerald Kaefer, Jacky Keong, and Anna Liu. Guest Editors' Introduction: Software Engineering for the Cloud. IEEE Software, 29:26–29, 2012.

[8] W. Hasselbring and R. Reussner. Toward trustworthy software systems. IEEE Computer, 39(4):91–92, April 2006.

[9] Nina S. Marwede, Matthias Rohr, André van Hoorn, and Wilhelm Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09), pages 47–57. IEEE, 2009.

[10] S. Pearson and Marco Casassa Mont. Sticky policies: An approach for managing privacy across multiple parties. IEEE Computer, 44(9):60–68, 2011.

[11] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. The Common Component Modelling Example (CoCoME), volume 5153 of Lecture Notes in Computer Science. Springer Verlag Berlin Heidelberg, 2011.

[12] Matthias Rohr, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In Claus Pahl, editor, Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08), pages 80–85, February 2008.

[13] André van Hoorn. Adaptive capacity management for the resource-efficient operation of component-based software systems. In Felix C. Freiling, Irene Eusgeld, and Ralf Reussner, editors, Proceedings of the 2008 Dependability Metrics Research Workshop, Technical Report TR-2009-002, pages 7–11. Department of Computer Science, University of Mannheim, Germany, May 2009.

[14] André van Hoorn, Holger Knoche, Wolfgang Goerigk, and Wilhelm Hasselbring. Model-driven instrumentation for dynamic analysis of legacy software systems. In Proceedings of the 13th Workshop Software-Reengineering (WSR 2011), pages 26–27, May 2011. (Softwaretechnik-Trends 31(2) (May 2011) 18–19).

[15] André van Hoorn, Matthias Rohr, Asad Gul, and Wilhelm Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In Nenad Medvidovic and Tetsuo Tamai, editors, Proceedings of the 2nd Warm-Up Workshop for ACM/IEEE ICSE 2010 (WUP '09), pages 41–44. ACM, April 2009.

[16] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, Germany, November 2009.

[17] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012), pages 247–248. ACM, April 2012.