Automatic Conformance Checking for Migrating Software Systems to Cloud Infrastructures and Platforms

Sören Frey, Wilhelm Hasselbring, and Benjamin Schnoor
Software Engineering Group, Kiel University, Kiel, Germany

May 26, 2011

This is the pre-peer reviewed version of the following article:


which has been published in final form at

http://dx.doi.org/10.1002/smr.582
Automatic Conformance Checking for Migrating Software Systems to Cloud Infrastructures and Platforms

Sören Frey, Wilhelm Hasselbring and Benjamin Schnoor

1 Software Engineering Group, University of Kiel, 24118 Kiel, Germany

SUMMARY

The migration of software systems to IaaS or PaaS-based cloud environments enables SaaS providers to benefit from the cloud’s merits, as for example smoothly scaling up and down existing applications. Our approach CloudMIG aims at supporting SaaS providers to perform those migrations. Here, validating the specific constraints that are imposed by a cloud environment constitutes an important early phase activity. For example, the access to the underlying file system, the number of certain file types, or calls to specific methods may be restricted by cloud providers. Those constraints have to be considered when evaluating the suitability of competing cloud environment candidates. We describe a generic cloud environment model that incorporates these constraints and present automatic violation detection mechanisms. A software system’s conformance can be examined with the assistance of constraint validators. They operate on extracted KDM-based system models and can, among others, apply metrics modeled with SMM through our metrics execution engine (MEE). Additional constraint validators can be plugged into the validation process as needed. In this context, we implemented a prototype and modeled the PaaS environment Google App Engine for Java. We report on a quantitative evaluation regarding the detected constraint violations of five open source systems. Copyright © 2011 John Wiley & Sons, Ltd.

Received …

KEY WORDS: Cloud computing; Migration to the cloud; Cloud environment constraints; CloudMIG; KDM; SMM

1. INTRODUCTION

Adopting technologies from the field of cloud computing [1–4] is a worthwhile option for many companies considering to modernize existing applications, whole application landscapes, and the accompanying procurement, operation, and maintenance processes. Among others, improved scalability, reliability, and the often-cited cut of capex costs constitute veritable stimuli. Furthermore, cloud computing technologies can facilitate the concentration on core business activities for software service providers. This is valid for in-house providers as well. Software service providers are called SaaS providers in the cloud computing context [5]. We investigate obstacles the SaaS providers face when migrating existing applications to the cloud and thereby build upon infrastructures and platforms delivered as services from IaaS and PaaS cloud providers. While the potential advantages accompanied with a migration might be compelling, there exist numerous difficulties SaaS providers have to overcome before being able to utilize cloud technologies and particularly to leverage a cloud’s capabilities. A cloud environment may enable superior scalability, but many legacy applications’ architectures are not designed to exploit the
provided elasticity, for instance. Therefore, it is often necessary to restructure an architecture to take full advantage of cloud technologies. But even running an unimproved existing application in the cloud while striving after a minimal change set can be a challenging task. Our model-based approach CloudMIG [6] aims at supporting SaaS providers to evaluate different cloud environment candidates from a technical perspective and to assist reengineers in accomplishing restructuring activities during the migration process. For being able to compare competing cloud environment offers, one essential aspect is to take adequacy for the specific legacy architecture and implementation into account. Not every cloud environment is similarly suited for a software system as the cloud environments impose varying constraints on the applications they host. For example, the ability to access the underlying file system may be permitted in a non-uniform way, or the usage of particular network protocols might be restricted. Hence, for each cloud environment being under consideration a detailed analysis has to be performed to validate the specific constraints.

We present an extensible architecture for modeling those cloud environment constraints (CECs) and detecting cloud environment constraint violations (CEC violations) to automate the conformance checking process. Thus, the specific CECs for a cloud provider can be documented in a reusable manner in a so called cloud profile and serve as a validation input for arbitrary legacy systems. The conformance of a software system with the modeled CECs can be examined with the help of constraint validators. Each constraint validator can check an existing or reconstructed model of the software system for code artifacts that would lead to CEC violations when being deployed unmodified. The validation is realized for software system models that are based on the Knowledge Discovery Meta-Model† (KDM). However, our architecture conceptually allows to widen the scope and provides extension points to smoothly incorporate other meta-models as well. Every constraint validator refers to a certain constraint type. For cases where the standard constraint validators might not yield sufficient results or a more efficient validation processing can be applied, the default set of validators can be extended. To ease the provision of detection mechanisms in such cases, additional constraint validators can be plugged into the overall validation process. For generic evaluation of KDM-based models we provide, among others, a constraint validator that can employ metrics that are formulated using the Software Metrics Meta-Model‡ (SMM). These metrics are computed by our metrics execution engine (MEE). Generally, the validation process builds upon a generic cloud environment model (CEM). Besides the model elements for describing the CECs, the CEM contains further aspects relevant for providing migration support. Among those are elements for modeling the structure of specific cloud environments, legacy code data containers, or virtualized hardware resources, for instance. We further studied the approach’s properties by modeling the PaaS environment Google App Engine for Java V.1.3.6 including the identified CECs and implemented the prototype tool CloudMIG Xpress that can execute the validation process. We report on a quantitative evaluation where we used the architectural models of five open source systems that were extracted by a model discoverer tool and investigated their intrinsic CEC violations by applying the validation process. This article is an extended and updated version of the work presented in [7]. Besides numerous improvements, we primarily added and evaluated the following two further key points. First, we describe important aspects for prioritizing detected CEC violations and accordingly propose a partitioning function. Here, information contained in the CEM is combined with the expected complexity for reworking that results from the affected classes’ coupling. Second, the usefulness of SMM in the scenario of migrating applications to the cloud is demonstrated. The measurements provided by MEE are utilized in the constraint validation process, for instance.

The remainder of the paper is structured as follows: Section 2 summarizes key aspects when migrating software systems to the cloud and focuses particularly on CECs. An overview regarding the metrics execution engine MEE is given in Section 3. In Section 4, we describe the extensible architecture for detecting CEC violations that includes the constraint validators and our cloud environment model CEM. Section 5 presents the quantitative evaluation. The related work is discussed in Section 6 before Section 7 draws the conclusions.

2. MIGRATION TO THE CLOUD

2.1. Challenges

SaaS providers have to overcome numerous challenges and shortcomings of current approaches when migrating existing software systems to the cloud. Organizational implications might include a reshaping of internal divisions as responsibilities of IT or software maintenance departments shift, for instance. In addition, new liability or auditing issues may arise because sensible data is no longer stored exclusively on premise. Along with the problem to identify data assets that can be moved to the cloud comes the increased need to encrypt this data. However, recent advancements in the cryptography domain in achieving fully homomorphic encryption might eventually enable practicable arbitrary computation on encrypted data without the necessity to decrypt it beforehand and therefore mitigate data security concerns to a great extent [8].

Analyzing technical challenges of a migration we identified the following major shortcomings of current approaches [6]: Solutions for migrating software systems to the cloud are limited to particular cloud providers (1). A poor level of automation concerning the creation of a target architecture, a mapping to this architecture, and the detection of CEC violations is prevalent (2). Further on, resource efficiency is not taken into account sufficiently. This is becoming particularly relevant regarding the prevailing pay-per-use billing models (3). Finally, automatic support for evaluating a target architecture’s scalability at design time is rare in the cloud computing context (4). These shortcomings constitute core drivers in the design of our overall approach CloudMIG that is outlined in the next Section 2.2 to explain the context. The definition of CEM and the quantitative evaluation regarding detected CEC violations presented in this paper address the shortcoming (1) and the last part of (2).

2.2. The Approach CloudMIG

CloudMIG is our approach for supporting reengineers to semi-automatically migrate existing software systems to cloud-based applications. It incorporates usage patterns and varying resource demands in creating a target architecture candidate and concentrates on enterprise software systems offered by SaaS providers. The approach is composed of six major activities that are illustrated in Fig. 1 (see [6]). The activities are briefly described in the following.

A1 Extraction: Includes the extraction of architectural and utilization models of the legacy system. They base upon OMG’s Architecture-Driven Modernization\textsuperscript{5} efforts and utilize KDM and SMM.

A2 Selection: Selection of an appropriate CEM-compatible cloud profile candidate.

A3 Generation: The generation activity produces the target architecture and a mapping model. Thereby, a model describing the target architecture’s CEC violations is created that serves as an input artefact for the target architecture generation process. The violations are detected with the aid of the extensible architecture described in this paper.

A4 Adaptation: The adaptation activity enables a reengineer to manually adjust the target architecture.

A5 Evaluation: The evaluation activity involves static analyses and a runtime simulation of the target architecture.

A6 Transformation: The actual manual transformation of the existing system from the generated target architecture to the aimed cloud environment.

\textsuperscript{5}OMG ADM: \url{http://adm.omg.org/} [26 May 2011]
2.3. Cloud Environment Constraints

2.3.1. Definition  To clarify concepts, we now explain the constraint-related key terms that are used in this paper.

Cloud environment constraint (CEC): A constraint imposed by a cloud environment related to a specific technical action. When a (potential) guest application attempts to execute this specific action, the cloud environment prevents it from being executed.

Cloud environment constraint violation (CEC violation): Action of a (potential) guest application that would be prevented from being executed by a cloud environment due to a related CEC. The CEC violation is caused by an action that is manifested in a source code element of the guest application.

Violation Severity: The severity of a CEC violation indicates the likely effort to fix the CEC violation during a migration process. As this can vary widely depending on specifics of the different legacy applications, we apply the violation severity rather pessimistic and propose three simple concrete severities: Breaking, Critical, and Warning associated with high, medium, and low effort, respectively.

It should be noted that the violation severity is biased according to the experience and subjective appraisement of a person modeling a cloud environment. Therefore, it should be seen as a hint for the reengineer. It is important to detect a CEC violation at all and to make the reengineer beware of it. Nonetheless, beyond that the violation severities of detected CEC violations can be used as an influencing factor when prioritizing the handling of CEC violations (see Section 2.3.4).

2.3.2. Examples  We give three examples of CECs, their associated violation severities, and properties of software systems that would lead to corresponding CEC violations. In our examples, we utilize constraints from our cloud profiles describing the Google App Engine for Java and Amazon EC2 cloud environments.
Figure 2. The possible states of a legacy software system for a specific target cloud environment in *CloudMIG* according to the validation of its CECs.

**CEC:** Using Google App Engine for Java, the total number of files is limited to 3,000 per default.

**CEC violation:** An application that exceeds this limit.

**Violation severity:** Warning (assuming that the creation of new container structures is a rather simple problem).

**CEC:** Using Amazon EC2, the local storage of VM instances is transient. For persistent storing one of Amazon’s services like EBS or RDS has to be used.

**CEC violation:** An application that writes to the local file system in one of its methods.

**Violation severity:** Critical.

**CEC:** Using Google App Engine for Java, only JVM-compatible languages can be used for guest applications.

**CEC violation:** A C++ application.

**Violation severity:** Breaking.

2.3.3. Management with *CloudMIG* CECs are made explicit in *CloudMIG*. Cloud profiles are modeled according to the CEM (see Section 4.1) and describe a specific cloud environment in a reusable manner. The cloud profiles contain specifications of the CECs which the cloud environment establishes. Furthermore, *CloudMIG* provides means for automatically detecting CEC violations of a legacy application and points the reengineer to architectural elements that cause the violations. These detection mechanisms are designed to be extensible for cases where the standard constraint validators may not be sufficient.
On the one hand, the detected **CEC violations** can be utilized by the reengineer to obtain a quick overview of problematic system parts which need special attention and as a basis for comparing competing cloud environment offers. On the other hand, the capability to detect **CEC violations** is used in the adaptation and evaluation activities (A4 and A5) to reason about the quality of different target architectures. As the reengineer has the possibility to manually adjust a target architecture, the detected **CEC violations** can vary over time and are therefore computed more than once.

Fig. 2 shows the integration of **CECs** in **CloudMIG**’s workflow. Before the initial validation of the **CECs**, a legacy software system is marked as “unchecked.” Afterwards, further proceeding depends on the existence of **CEC violations** and the **violation severities** of detected **CEC violations**. If **Breaking** violations exist, the legacy system is considered as “incompatible.” Referring to example three from Section 2.3.2 again, a migration would imply a transformation from one programming language into another. **CloudMIG** does not provide further support for such kinds of migrations and therefore the workflow ends in the case that **Breaking** violations exist. Otherwise, the legacy system is either regarded as “compatible” (no **CEC violations** exist) or “pending” (**Critical** or **Warning** violations exist). The distinction is made as for “pending” legacy systems feedback for the reengineer and tracing to the sources of the **CEC violations** has to be provided, for example. **CloudMIG**’s activities A3–A5 are executed subsequently in both cases. As mentioned above, the detected **CEC violations** can vary over time and therefore both states “Aligned (Clean)” (no **CEC violations** exist) and “Aligned (Dirty)” (**Critical** or **Warning** violations exist) can be reached along both paths after final constraint validation. Accomplishing the actual transformation ends the migration from **CloudMIG**’s perspective.

The **CECs** are analyzed in **CloudMIG** with so called constraint validators. As with components for most of the approach’s activities, they can be plugged into the architecture, too. The components presented in Fig. 3 exhibit their affiliation to **CloudMIG**’s activities through related activity numbers.

### 2.3.4. Prioritization of CEC Violations

When deciding for a target cloud candidate after validating the **CEC violations** for the existing software system, the detected **CEC violations** have to be fixed manually in **CloudMIG**’s activity A6. All of these **CEC violations** have to be addressed to enable proper operation. Nonetheless, reasoning about a suitable order has the potential to improve the actual transformation towards the intended target architecture. The observed **violation severities** can be used as a first indicator to partition the **CEC violations**. Here, the **CEC violations** with higher **violation severities** should be handled primarily because by tendency they imply more uncertainties regarding the overall development effort. However, the **violation severities** are defined generically in a model describing the target cloud environment itself (see Section 4) and therefore can not take the existing system’s characteristics into account. Referring to this, other important influencing factors that are unique to that system can be seen in the type and extent of consequences for other system parts once tackling a particular **CEC violation**, as well as the specific complexity of the element exhibiting the **CEC violation** caused by numerous dependencies. These factors can
be covered through incorporating a type-level coupling metric. Furthermore, the CEC violations should be prioritized and grouped reflecting manageable existing boundaries to facilitate appropriate assignment to developers, for instance. Hence, we partition the CEC violations along classes and consider their coupling values as well as the involved violation severities as mentioned before.

A system model contains a set of classes that we call $\tilde{C}$. We denote the coupling of a class $c \in \tilde{C}$ as $coup(c)$ and define $\alpha(c)$ as the number of the class’ CEC violations to the total number of the system’s CEC violations ratio. CloudMIG solely provides support for migrating software systems to cloud environments where no Breaking violations are detected (see Section 2.3.3). Therefore, we further distinguish merely Critical from Warning violations and define $\gamma(c)$ as the number of the class’ Critical violations to the total number of the system’s Critical violations ratio. $\alpha(c)$ and $\gamma(c)$ have to be considered in combination when reasoning about a prioritization of CEC violations. For example, a class exhibiting a large number of CEC violations where all violations are just Warning violations can have a similar impact to a class that reveals just a few CEC violations which are all of the Critical severity. Thus, we include both when prioritizing the CEC violations and as we regard them equally important we define the class violation weight (cvw) of class $c \in \tilde{C}$ as follows:

$$cvw(c) = \alpha(c) + \gamma(c)$$

The final prioritization procedure that is utilized later in our quantitative evaluation also incorporates the coupling influence factor and produces three partitions matching the priority levels I–III. Here, priority level I corresponds to the most and priority level III to the least urgent categories for classes raising CEC violations, respectively. Denoting $mean(coup(\tilde{C}))$ as the mean of the coupling values of all classes in the system and $mean(cvw(\tilde{C}))$ as the mean of the class violation weights of all classes in the system, we define the prioritization function $prio$ that assigns a priority level to a class $c \in \tilde{C}$ in the following way:

$$prio(c) = \begin{cases} 
I & \text{, } coup(c) > mean(coup(\tilde{C})) \land cvw(c) > mean(cvw(\tilde{C})) \\
II & \text{, } else \\
III & \text{, } coup(c) \leq mean(coup(\tilde{C})) \land cvw(c) \leq mean(cvw(\tilde{C}))
\end{cases}$$

The prioritization function $prio$ is defined for a class $c \in \tilde{C}$ that raises at least one Critical or Warning violation. As described before, classes included in constellations (a software system evaluated with respect to a specific cloud environment) exhibiting Breaking violations or classes without any CEC violation need not to be prioritized for rework.

### 3. THE METRICS EXECUTION ENGINE

The metrics execution engine $MEE$ applies metrics that are modeled with SMM to KDM instances. For example, we utilize $MEE$ during the constraint validation process or for the prioritization of CEC violations. In this Section, we first describe how to create SMM instances in CloudMIG Xpress and we briefly illustrate the general structure of such instances. Then, we introduce $MEE$’s measurement process and finally, we show two examples of SMM instances that are revisited in later sections.

#### 3.1. The SMM-Editor

At the time of this writing, SMM recently emerged as a novel promising technology to describe metrics in a generic fashion. Therefore, no comprehensive library of metrics does exist for the publicly available V. 1.0 Beta 1 specification at the moment. Hence, it is beneficial to provide convenient means for creating valid SMM instances. CloudMIG Xpress comes with an editor...
based on the Eclipse Modeling Framework (EMF)\footnote{http://www.eclipse.org/modeling/emf/} which allows to define arbitrary SMM-conform metrics. Fig. 4 shows the user interface which includes a tree-based editor and a property view. As indicated by the second topmost tab, for each SMM instance the editor additionally provides a helper class skeleton written in Java which can be extended in order to support more complex operations for the measures. It should be noted that in SMM the terms “measure” and “metric” are used as synonyms.

A valid SMM instance consists of a \textbf{Characteristic} which defines a common trait of the referenced measures. There must exist at least one measure which relates to the characteristic. The SMM specification provides different kinds of measures. We sketch important classes with the help of Fig. 5, whereas the semantics of the declared file count measure is elucidated in Section 3.3. Fig. 5 shows three different SMM measure types. The \textbf{BinaryMeasure} applies the defined functor operation to the results given by its both base measures. Furthermore, an \textbf{AdditiveMeasure} is defined which accumulates the results given by its single base measure. Finally, there is a \textbf{Counting} measure defined which applies a stated operation directly to KDM elements. Each measure relates to one \textbf{Scope} that defines the class type to which the measure can be applied. For example, Fig. 5 includes a \textbf{Scope} element having its \textit{class} attribute set to \texttt{source::InventoryModel}. Thus, the associated binary measure can only be applied to \texttt{InventoryModel} elements of KDM instances. Finally, an SMM instance can contain measurements. Those elements are the results of metric computations and will be attached automatically by \texttt{MEE} during the measurement process.
3.2. The Measurement Process

The measurement process of MEE applies SMM measures to KDM-based models. It can be divided into the following three phases.

1. Initialization
2. Recursive measure application
3. Assignment

During the initialization phase, MEE has to discover several elements as a start. First, it determines the entry point of the overall defined metric. Therefore, it has to find the measure which is not a base measure for any other measure defined in the current context. The modeler of an SMM instance has to ensure that there exists exactly one measure which fulfills this condition. In the following, we will call this measure the main measure. After the engine has found the main measure, it discovers the KDM instances for all types that match the main measure’s scope because this measure has to be applied to all of them separately. Afterwards, the second phase starts and the main measure along with its associated measures will be applied to the KDM instances discovered in the preceding phase. Further proceeding depends on the measures’ types. For example, a single utilization of Counting applied directly on a KDM instance element produces a single measurement, whereas the AdditiveMeasure results in computing multiple measurements and the according summing up (see Section 3.3). As mentioned above, the measurement results will be attached to the SMM instance. This is accomplished in the last phase. Additionally, some meta information such as the execution date is added and the mapping between the measurement elements and the measurands is established.

3.3. Examples

In this Section, we outline two metrics that are modeled with the help of two SMM instances. Fig. 5 shows an SMM instance which can be used to count the number of files. Hence, the characteristic

Figure 5. SMM file count measure variant to compute the number of files omitting source files.
is named "File count". As we utilize it later in a constraint validator that considers only files that are being deployed to production systems (see Section 4.1.3), this SMM instance omits source files. Generally, the KDM instances contain, among other models, an InventoryModel. All files related to the software system are described in such a model. The BinaryMeasure will be applied by MEE as a main measure because it is not a base measure for any other measure. The engine searches for an InventoryModel. If it discovers one, it will first apply the baseMeasure1 to all InventoryItems contained in the model. The result will be the number of all files included in the software system model. Afterwards, the engine applies the baseMeasure2 to all InventoryItems. The resulting measurement represents the number of all source files contained in the KDM model. Finally, the engine subtracts the number of source files from the number of all files. Thus, the result will be the number of all files except the source files.

Fig. 6 shows a simple type-level coupling metric. The SMM instance can be used to determine the number of imports for a class. Though, it will only count the references to internal classes. This measure counts the number of import statements for each class of the software system. The main measure of this SMM instance is the AdditiveMeasure that will be applied to ClassUnit elements (classes) of KDM instances. If the engine discovers such an element, the operation of the Counting measure will be applied to all Imports that are contained in the class. The result of the operation that is implemented in the helper class will be 1 if the class referenced by the import statement is defined in the software system. It will return 0 if the class is included in external libraries. Hence, after the AdditiveMeasure sums up the according measurements, the result will be the number of referenced internal classes for a single class. Due to the definition in the SMM instance, this metric will be computed for every class in the software system.

4. THE ARCHITECTURE FOR DETECTING CEC VIOLATIONS

4.1. Cloud Environment Model

The Cloud Environment Model (CEM) constitutes the foundation for CloudMIG’s capabilities to detect CEC violations. It is aligned with OMG’s KDM and comprises a model for describing CECs. Instances of CEM represent specific cloud environments (cloud profiles) and have to be modeled only once and can then be reused by other reengineers. This applies to included CECs and already implemented constraint validators, too. It is intended to provide an appropriate public repository in the future.

4.1.1. Overview The CEM is organized in layered packages as presented in Fig. 7. The Core package includes basic elements like abstract cloud services or partitions. The last allows to model both Amazon EC2’s Availability Zones and Regions, for instance. The further packages build
upon the core package. The Mapping package comprises model elements that enable integration of legacy system parts into a cloud environment. “Mapping” therefore means the assignment of legacy system parts to entities available in the cloud domain. Potential incompatibilities are handled by means of adapters that have to be created manually in the subsequent transformation step (CloudMIG activity A6), for instance. The Usage Package contributes model elements for describing and extracting the utilization model used by CloudMIG. In doing so, it incorporates measures and measurements modeled with SMM. The Constraints package models the CECs and is covered in Section 4.1.3 in greater detail. The IaaS package and PaaS package comprise elements for the corresponding cloud service models. The first follows the structural elements of the cross platform cloud API Deltacloud\textsuperscript{\parallel} to some degree. The last is designed more generic as PaaS clouds exhibit an even broader bandwidth. The Cloud Profile package forms the entry point for modeling specific cloud environments. An excerpt is presented in Fig. 8. A CloudEnvironment can contain several CloudEnvironmentConfigurations. Taking Google App Engine as an example, there exists a CloudEnvironmentConfiguration for Google App Engine for Java and one for Google App Engine for Python. Furthermore, Fig. 8 shows elements for including an IaaS HardwareConfiguration (e.g., an Amazon EC2 “High-Memory Extra Large Instance”), an EnvironmentConstraintConfiguration for incorporating CECs, and the abstract classes AbstractCloudService and AbstractCloudAppDataContainer for providing convenient generic extension points for cases where the concrete instances in other layers may not be sufficient. The alignment with KDM is described in the following Section.

4.1.2. Alignment with KDM Besides domain-specific elements of the Cloud Profile package, one can recognize the alignment of CEM with KDM in Fig. 8. Generally, CEM builds upon KDM’s platform and structure packages following the piggyback pattern \cite{9} for the realization of DSLs. We provide a transformation from the domain model implemented as an Ecore model to a KDM-compatible version. Future KDM-conform modernization tools may therefore be able to process our model. The compatibility is achieved by avoiding to introduce new meta model elements. The CEM classes that inherit from KDM classes (gray) in Fig. 8 are rather transformed to KDM Stereotypes. Their attributes become KDM TagDefinitions, and the CEM packages are realized as KDM ExtensionFamilies, to cover just the major modeling elements. This approach uses the light-weight extension mechanism of KDM. The mentioned elements constitute new so called virtual meta-model elements. They have to consider given restrictions. For example, the CloudEnvironmentConfiguration inherits from the KDM type PlatformElement and therefore can not incorporate the EnvironmentConstraintConfiguration class.

\textsuperscript{\parallel}\texttt{http://www.deltacloud.org/} [26 May 2011]
4.1.3. Modeling of Constraints

CEM's EnvironmentConstraintConfiguration class was already mentioned in the former Section 4.1.2. It is located in the CEM Constraints package. An excerpt of this package is presented in Fig. 10. An EnvironmentConstraintConfiguration constitutes a container for a set of CECs. Concrete CECs are modeled by inheriting from AbstractConstraint which provides an attribute for a ViolationSeverity, for instance. In Fig. 10, some concrete CEC examples are listed for illustrating purposes, as the FilesystemAccessConstraint for detecting a forbidden file system access, or the ReflectionConstraint for discovering a prohibited call to the reflection API, for example. The BasicOCLConstraint allows a generic detection on MOF-compliant models like KDM. Instances of TypeList provide a convenient way to refer to a predefined set of types, or to create an own set that can be used for modeling CECs. For example, CloudMIG Xpress contains a predefined set of qualified JRE 6 types that can be applied in Google App Engine for Java's

**The KDM Ecore model used from the tool MoDisco names the “extensionFamily” role of the according containment relationship in the “KDMFramework” super class element merely “extension”**

Copyright © 2011 John Wiley & Sons, Ltd.


DOI: 10.1002/smr
case in a TypesWhitelistConstraint as a superset (ClosureContains relationship), as Google App Engine for Java restricts access to only a subset of these types (TypeListContains relationship). The SMMConstraint allows to use dimensional metrics formulated with SMM to model constraints. Those types of constraints employ our metrics execution engine MEE to compute the metrics. There exist further subclasses of the SMMConstraint. For example, the MaxTotalNrOfFilesConstraint asserts file limits and is implemented employing the according SMM model that is outlined in Section 3.3.

### 4.2. Constraint Validation

**CEC violations** are detected by validating the CECs that are modeled in a cloud profile with regard to the reconstructed architectural model that results from CloudMIG’s activity A1, for instance. Classes playing a role in the CEC validation process are shown in Fig. 11. Constraint validation plugins for CloudMIG Xpress (see Fig. 3) have to provide a subclass of AbstractConstraintValidator. Currently, for reasons of convenience there exist three further abstract subclasses one can inherit from, namely variants for static validation (AbstractStaticConstraintValidator), for static validation with respect to KDM source models (AbstractConstraintKDMValidator), and for incorporating metrics that are modeled with SMM and are applicable to KDM-based models (AbstractConstraintSMMKDMValidator). Constraint validators refer to a specific constraint type, for a given CEC there may exist various constraint validators. An AbstractCommonTypeListManager handles the predefined lists of types mentioned in Section 4.1.3.

The CEC validation process basically functions as follows. CloudMIG Xpress queries every present constraint validator whether the programming language of the source model is supported (isSuitedFor–method) and if the offered validation capability matches a constraint type present in a cloud profile (getConstraintTypeName–method). It should be noted that in the case for language independent validation plugins the first method always returns true. If an appropriate constraint validator is found, CloudMIG Xpress initializes it and launches the specific validation process with a call to the method initialize and validate, respectively. In the case that CEC violations are detected, the constraint validator returns a list of
AbstractConstraintViolations due to a call to the getViolations method after the plugin finished validation.

5. EVALUATION

To study characteristics of detecting CEC violations and to achieve insights in quantity, types, and properties of CEC violations a reengineer might face when considering a migration of an existing system to the cloud, we evaluated our approach by means of five open source systems and Google App Engine for Java\textsuperscript{††} V. 1.3.6. Our prototype tool CloudMIG Xpress was utilized to process the extracted models of these systems and to perform the validation process to detect CEC violations. For each detected CEC violation a data record was reported that comprised the IDs of the processed application, the particular KDM model, and the class referring to the source of the CEC violation, as well as the corresponding violation severity and the type of the constraint.

\textsuperscript{††}http://code.google.com/appengine/docs/java/overview.html [26 May 2011]
5.1. Background

We modeled the PaaS Cloud Environment Google App Engine for Java to evaluate our approach. Accordingly, we created a CEM instance comprising structural elements and the corresponding CECs. In CEM’s terms, Google App Engine for Java constitutes a CloudEnvironmentConfiguration as part of the CloudEnvironment Google App Engine, as mentioned in Section 4.1.1 before. It allows JVM-compatible languages to be used by guest applications and provides them an isolated sandbox environment with a considerable number of restrictions. Despite that not all restrictions are documented extensively, they form a sufficiently well-suited basis for modeling the related CECs. The information concerning functionality, structure, and sandbox restrictions were distilled from Google’s webpage and several further web logs provided helpful information for understanding more facets of the specific CECs.

In the context of this evaluation, an important aspect was judging about the feasibility of our approach and less exploring its applicability on a wide range of diverging system types. Therefore, we narrowed down the type of potential applications to Java and web-based systems. The open source systems listed in table I were selected for evaluation purposes. Some basic characteristics of each system can be found in table II. Considering LOC of the application’s sources while leaving out used third-party libraries, the sizes of the largest and the smallest application differ by a factor of approximately 33 and therefore allow analyses for varying scales.

5.2. Methodology

For extracting the related KDM models of the applications, we applied the tool MoDisco [10] V.0.8. In our analysis, we distinguished cases where we solely considered the application’s own sources and those taking into account third-party libraries. This is due to the fact that some needed elements...
Table I. The open source systems utilized in the evaluation.

<table>
<thead>
<tr>
<th>App</th>
<th>Name</th>
<th>URL [26 May 2011]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Coefficient Core 0.9.6</td>
<td><a href="http://coefficient.sourceforge.net/">http://coefficient.sourceforge.net/</a></td>
</tr>
<tr>
<td>2</td>
<td>iBATIS JPetStore 4.0.5</td>
<td><a href="http://ibatisjpetstore.sourceforge.net/">http://ibatisjpetstore.sourceforge.net/</a></td>
</tr>
<tr>
<td>3</td>
<td>JavaBB 0.99</td>
<td><a href="http://www.javabb.org/">http://www.javabb.org/</a></td>
</tr>
<tr>
<td>4</td>
<td>jForum 2.1.9</td>
<td><a href="http://jforum.net/">http://jforum.net/</a></td>
</tr>
<tr>
<td>5</td>
<td>Ace Operator 1.7.0</td>
<td><a href="http://www.quik-j.com/ace_operator.htm">http://www.quik-j.com/ace_operator.htm</a></td>
</tr>
</tbody>
</table>

Table II. Basic characteristics of the open source systems utilized in the evaluation.

<table>
<thead>
<tr>
<th>App</th>
<th>Domain</th>
<th>#Classes (w/o libs)</th>
<th>#Libraries</th>
<th>LOC (w/o libs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Collaboration platform</td>
<td>131</td>
<td>41</td>
<td>11,362</td>
</tr>
<tr>
<td>2</td>
<td>Pet store</td>
<td>43</td>
<td>12</td>
<td>2,132</td>
</tr>
<tr>
<td>3</td>
<td>Forum software</td>
<td>256</td>
<td>43</td>
<td>12,239</td>
</tr>
<tr>
<td>4</td>
<td>Forum software</td>
<td>316</td>
<td>30</td>
<td>29,563</td>
</tr>
<tr>
<td>5</td>
<td>Live support</td>
<td>556</td>
<td>26</td>
<td>69,516</td>
</tr>
</tbody>
</table>

Table III. The incorporated CECs.

<table>
<thead>
<tr>
<th>Type</th>
<th>#Variants</th>
<th>Violation severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxTotalNrOfFilesConstraint</td>
<td>1</td>
<td>1 Warning</td>
</tr>
<tr>
<td>MethodCallConstraint</td>
<td>12</td>
<td>3 Critical, 9 Warning</td>
</tr>
<tr>
<td>SocketOpeningConstraint</td>
<td>1</td>
<td>1 Critical</td>
</tr>
<tr>
<td>FilesystemAccessConstraint</td>
<td>2</td>
<td>1 Critical, 1 Warning</td>
</tr>
<tr>
<td>ReflectionConstraint</td>
<td>1</td>
<td>1 Critical</td>
</tr>
<tr>
<td>TypesInstantiationConstraint</td>
<td>1</td>
<td>1 Critical</td>
</tr>
<tr>
<td>TypesWhitelistConstraint</td>
<td>1</td>
<td>1 Critical</td>
</tr>
<tr>
<td>LanguageConstraint</td>
<td>1</td>
<td>1 Breaking</td>
</tr>
</tbody>
</table>

that are necessary for certain constraint validators were not present in the KDM models for the libraries discovered with MoDisco. For example, the models lacked instances of the class Calls that represents method calls among others. 29 CECs were modeled, it was possible to detect 20 CECs that could be covered by our provided constraint validators corresponding to 8 covered types of CECs. The lack of the residual CECs is caused by the fact that CloudMIG Xpress does, at the moment, not provide support for detecting violations that are only identifiable at runtime and just static analyses are implemented.

Table III lists the types of CECs that could be detected during the evaluation and the nr. of present variants. For example, several variants of a MethodCallConstraint exist due to restrictions on calls to different methods. It should be noted that for different variants of a CEC type there can be dissimilar violation severities assigned. Furthermore, the CECs inheriting from AbstractTypeListConstraint show only a single variant in the table. But looking at a TypesWhitelistConstraint, the single constraint translates to a prohibition to access 2,388 of the JRE types, for example. We examined the following questions of special interest:

**Q1** How many classes of an application raise CEC violations?  
**Q2** How does the density of CEC violations vary among applications?  
**Q3** What types of CEC violations are prevalent?  
**Q4** Regarding classes that raise CEC violations, how does the size of such classes relate to the number of CEC violations they raise?  
**Q5** How does the density of CEC violations vary among classes?  
**Q6** What partitioning is yielded when applying the prioritization function $\text{prio}$ to the discovered CEC violations?  
**Q7** How does the number of CEC violations raised by an application’s own sources relate to the number of CEC violations raised by utilized third-party libraries?
5.3. Results

The absolute numbers of detected CEC violations differ widely. For App1-App5, we detected (87/4,386), (3/8), (98/932), (273/1,428), (7,795/612) CEC violations (own sources/ third-party libraries).

Q1: Fig. 12 shows the amount of classes causing CEC violations. The range spans from approx. 2.5% to 33% for the smallest (App2) and the largest application (App5), respectively. Generally, with application size comes a steady growth in the share of classes raising CEC violations, but without a uniform growth rate or pattern.

Q2: Fig. 13 shows the detected constraint violation densities per application. They are similar for App1, App3, and App4 considering Critical violation severities, despite App4 being roughly 3 times bigger than the others. The smallest application App2’s density is slightly lower, but the biggest application App5’s density exceeds the others in orders of magnitude. Generally, we conclude that the size can also be an indicator for higher densities of CEC violations, too. But for CEC violations with Warning as violation severity this statement is weaker.

Copyright © 2011 John Wiley & Sons, Ltd.
Prepared using smrauth.cls
DOI: 10.1002/smr
Q3: In Fig. 14, the distribution of detected violation types is presented. It is no surprise that the TypesWhitelistConstraint is dominant, as this CEC can match for a plethora of cases as stated in Section 5.2. APP2 does not coincide, but this is likely due to the low overall number of CEC violations found for this application. CEC violations related to restricted file system access are in addition to it quite frequently observed.

Q4: Fig. 15 shows a scatterplot for classes raising CEC violations and their relation to the relative number of CEC violations they raise and their size. The predominant number of those classes are responsible for 5% or less of the overall number of CEC violations raised. The root causes are wide spread over the systems. But there exist some outliers, the likely most spectacular one referring to App3 and being responsible for approx. 60% of CEC violations. All CEC violations from the small App2 are located in one class.

Q5: Fig. 16 shows the detected constraint violation densities per class. The largest application App5 exhibits the largest jitter. It is remarkable that the medians of all classes in all applications lie in a rather narrow band of approx. 3-5 (normalized to 100 LOC for each class).

Q6: Fig. 17 presents the partitioning that results from applying the prioritization function prio in combination with the import coupling measure described in Section 3.3. It should be noted that the application App2 is omitted in Fig. 17 due to legibility and moreover the CEC violations of App2 (considering solely its own sources) stem from a single class. Therefore, a prioritization at the class level would be rather pointless for App2. Analyzing the results for the other applications, we observe that priority level I is chosen in a range of 0% to 14% of the classes (App3 and App1). 26% to 44% of the classes (App5 and App4) are assigned priority level II and 42% to 66% of the classes (App1 and App5) receive priority level III. A pattern that we register for our test subjects is that the size of an application is correlated inversely to the share of classes that are assigned the priority level I. However, the correlation does not exhibit a constant factor and additionally App3 constitutes an exception lacking priority level I classes at all. Taking a closer look at the other
applications’ priority level \( I \) classes, we notice that the vast majority of them is bigger in terms of LOC than the average of the classes raising CEC violations.

**Q7:** In Fig. 18, the third-party libraries are incorporated. It presents the origin of CEC violations. Already the total numbers stated above showed vast differences. Once more, App5 is a special case, as it is the only system that produces substantially more CEC violations in its sources than its third-party libraries do. Furthermore, it can be seen in Fig. 18 a)-c) and partially in d) that few libraries are responsible for most of the CEC violations. Identifying and handling those primarily seems to be a worthwhile approach.

### 5.4. Discussion

The evaluation investigates CEC violations in a PaaS context. Compared to current IaaS environments, the individual PaaS offerings are more heterogeneous as they add diverse abstraction layers or specific preconfigured software stacks to the basic infrastructure building blocks. From a cloud user perspective, these additional boundaries constitute further CECs that need to be taken into account. Considering other PaaS environments, the results from the evaluation will primarily shift because of two factors. First, PaaS environments that do not support the Java runtime environment will yield incomparable results. As stated in Section 2.3.3, the approach CloudMIG does not aim to migrate software systems through applying a programming language transformation. It terminates its workflow if CEC violations with the assigned violation severity Breaking are detected. The LanguageConstraint listed in table III is a Breaking constraint. Second, the majority of detected CEC violations are TypesWhitelistConstraints. Hence, concordance of results for other PaaS environments offering a JRE compatibility will particularly be sensitive to type restrictions diverging from those defined in Google App Engine for Java’s sandbox environment.

Regarding IaaS environments, it can be expected that the number of detected violations would be considerably lower. This is due to the fact that IaaS environments by definition lack narrow restrictions concerning the underlying software stack that in turn could translate into additional modeled CECs and violations of those. Furthermore, identifying orphaned classes, components, or
even subsystems is a typical task in reengineering projects. Here, static and dynamic analyses can be used. Considering only system parts that are actually being used might also lead to a significant reduction of CEC violations. This is planned for future analyses.

The prioritization function $\text{prio}$ provides a partitioning of classes that are responsible for CEC violations rather than a strict partial ordering. It addresses two important factors that presumably have effects regarding the reengineering effort, that is to say the coupling [11, 12] and violation severities of classes. However, as there may be other factors not covered by $\text{prio}$, we solely provide coarse grained partitions indicating the urgency of reworking the classes. Here, higher priority levels exhibit a greater amount of uncertainty that, from a migration planning perspective, should be eliminated as early as possible. Furthermore, coupling metrics that are rather sophisticated could be employed as well. To integrate characteristics specific for each application, $\text{prio}$ does provide a relative partitioning rather than using fixed thresholds. Moreover, it combines the averages of the coupling and violation severity values as we are explicitly interested in finding outliers and do not aim for uniform distributions among priority levels. This construction can explain the observed continuous decrease in the number of priority level $I$ classes that is in most cases associated with increased sizes of the applications.

Furthermore, only system types were analyzed that were relatively well-suited. First, our approach assumes that KDM models can be extracted. This might not be the case for many existing programming languages as they currently lack appropriate tool support. As mentioned before, the extracted models from third-party libraries were also incomplete and therefore results for $Q7$ can rather show a tendency. However, the used tool MoDisco provides a suitable framework for developing and enhancing discoverers. Second, Google App Engine for Java supports JVM-based web software systems. Only representatives of this system type were studied and the number of probands was also rather small. While additional work is needed to investigate specifics regarding further system types and IaaS cloud environments to validate our findings, the experiments provided us valuable insights and showed that our approach can be successfully utilized. All detected CEC violations have to be either addressed in reengineering measures or at least be considered in

Figure 15. Classes being responsible for an amount of constraint violations and the relation to their size (w/o third-party libs). Split into two diagrams (a) and (b) due to legibility.
the decision-making process while evaluating target cloud environment candidates. As described before, the partitioning that results from applying the prioritization function $prio$ can serve as a guideline when reasoning about the order of reengineering the classes that raise CEC violations.

6. RELATED WORK

6.1. Context

Our approach is located in the domain of software migration. An overview of legacy software system migration was contributed in [13]. The migration of existing systems to new platforms shows inherent complexity and numerous difficulties a reengineer has to overcome and migrating legacy systems to cloud-based environments makes no difference at this point. The authors in [14] propose several techniques to reduce migration complexity, for example dynamic program analysis, software visualization, and knowledge discovery. Among other techniques, we utilize metrics modeled with SMM to validate CECs. The authors in [15] aim to provide consistent concepts and a terminology for software measurement. The vocabulary of SMM differs to some degree, but there are counterparts for the most important concepts that are relevant for our use case. Domain specific model metrics are addressed in [16] and the authors use SMM to measure models as well. Here, metrics can be formulated with OCL and are then translated to an SMM instance according to generation rules that are phrased in OCL, too. Their prototype tool Metrino is also able to compute the generated metrics. Compared to this approach, a reengineer may add additional metrics to CloudMIG Xpress that are formulated using SMM. Predefined or manually added SMM instances can be applied by MEE to KDM-based models. For example, MEE is utilized in the constraint validation process if corresponding CECs are defined in a cloud profile.
6.2. Cloud Computing

The novel field of cloud computing constitutes a major foundation of our work, too. The cloud computing paradigm as well as its different service models are described in [1–4], for instance. The authors in [17] provide an overview of 20 cloud definitions and extract a consensus definition. In [18], the research published in the area of cloud computing is surveyed and the lessons learned...
from related technologies are discussed to provide the foundations for shaping a cloud research agenda. Another approach for defining a research agenda for this field is described in [19]. For example, the authors highlight research topics like power management and the stability of large-scale event notification platforms. Further challenges in the cloud computing domain are examined in [20]. Amongst other subjects, legacy system integration issues and the challenges evaluating an application’s suitability are included here, too.

The CEM proposed in this work models arbitrary cloud environments including their CECs and provides a basis for considering an integration of an existing system through examining and mapping a reconstructed architectural model’s elements. Modelling cloud environments is investigated in [21] as well. Here, the authors suggest to use semantic web technologies like RDF and OWL. General requirements for a cloud computing system’s architecture are described in [22]. However, several of the included aspects that are relevant from a provider perspective when building a cloud computing architecture are less important in our context. For example, the user experience or the learning requirements from users are negligible when investigating technical migration issues.

6.3. Cloud Computing Adoption

Enterprises planning to apply cloud computing technologies to either complement on premise resources in a hybrid cloud scenario or to migrate entire software systems to the cloud face several challenges. As covered in this article, they need to detect the software systems’ CEC violations with respect to the potential target cloud environments, for instance. General cloud computing adoption issues, as factoring in security, interoperability, and compliance concerns, are described in [23] and [24]. A cloud adoption toolkit is proposed in [25]. It comprises a conceptual framework that is intended to support reasoning about incorporated concerns in a structured way and to match these concerns to appropriate tools. [26] aims to provide a cloud computing adoption strategy through building a roadmap called ROCCA (Roadmap for Cloud Computing Adoption). It is based on a survey investigating cloud computing adoption issues. For evaluating the compliance to the
roadmap, the ROCCA Achievement Framework (RAF) is proposed as well. The authors in [27] investigate the transformation into a cloud-based IT infrastructure with a focus on maintaining high performance characteristics. Therefore, they define a strategic framework that considers aspects like uncertainties regarding the performance when replicating applications in public clouds. A suitability index concerning the identification of a company’s suitability for the adoption of cloud computing is presented in [28]. It includes characteristics like the resources’ utilization patterns and the sensitivity of the incorporated data. The authors in [29] introduce a visual modeller tool for analyzing cloud computing environments and applications. However, no automatic mapping from a reconstructed software model to candidates of cloud-based target architectures is included. Moreover, the provided cloud environment model does not take existing CECs into account.

Considering migrations to the cloud, in [5] the effort emerging for those types of migrations is considered an important factor that needs to be taken into account before deciding for a move. Furthermore, it is shown that not each application is equally suited for running in the cloud. Whether a migration fulfills the expectations depends on the actual resource consumption patterns in combination with a specific pricing model offered by a cloud provider. Our approach CloudMIG addresses both concerns as it supports SaaS providers to migrate existing applications to the cloud and facilitates the evaluation of cloud environment candidates through employing automatic suitability analyses. A taxonomy of critical factors considering the migration of applications to the cloud is contributed in [30]. The authors conducted a number of sample migrations and categorized the involved difficulties and tasks, e.g. the understanding of the application environment and the selected cloud platform, that are both addressed by CloudMIG as well. The authors condense their findings in the constructed taxonomy. A related case study was conducted in [31]. The authors report on a migration that transferred an enterprise software system to an IaaS cloud environment. However, the case study concentrates on financial and socio-technical enterprise issues. [32] reports on a migration of a large scientific database to the cloud. The authors state problems transferring the data to the cloud due to the mere size of their database. Moreover, performance drawbacks were observed when following a least effort approach through minimizing changes regarding the database settings and schemata. [33] shows, that simplistic migration approaches are often not feasible in enterprise application scenarios, too. The authors highlight the inherent complexity of most enterprise software systems. A migration model is presented that takes enterprise-specific constraints into account, as for example influence on costs, increased transaction delays, or necessary adoption of security policies. The stated complexity of current enterprise applications is a driver for the authors of [34] as well. They propose an approach for reengineering enterprise software for cloud computing that is based on ontologies. For example, extracted models are transformed and integrated to an enterprise software ontology, that is partitioned to derive potential service candidates.

7. CONCLUSIONS

Cloud environments impose technical constraints that have to be considered when migrating software systems to the cloud, as for example limited access to the underlying file system or enforced restrictions regarding provided standard APIs. In this context, we presented an extensible architecture for detecting a software system’s violations against those constraints. The violation detection process and the highlighting of crucial system parts are essential early phase activities of our approach CloudMIG which supports reengineers in migrating existing software systems to the cloud. Constraint violations may be easy to fix in some cases, but a reengineer has to be aware of them without exception when evaluating cloud environment candidates. The extensible architecture includes a generic cloud environment model that incorporates these constraints, as well as reusable constraint validators to check a software system’s conformance with a specific cloud environment in terms of constraint violations. The constraint validators operate on extracted KDM-based system models and can, among others, apply metrics that are formulated using SMM through our metrics execution engine (MEE). Furthermore, our approach provides support for assessing the severity of a detected constraint violation. Moreover, we presented results concerning a quantitative
evaluation regarding the detected constraint violations of five open source systems. The applications were validated against the modeled PaaS cloud environment Google App Engine for Java. The following key insights were observed: With bigger LOC application sizes came (1) a steady growth in not only the total number of classes raising constraint violations but also in the ratio of those responsible classes, (2) often higher overall violation densities, and (3) a decreased ratio of those that were assigned high urgency priority level \( I \) when utilizing our prioritization function \( \text{prio} \) that intentionally narrowed the focus towards outliers. Furthermore, the responsible elements that raised constraint violations, as for example method calls or import statements, were often wide spread across the systems and were seldom condensed in bigger clusters. Additionally, the constraint violations’ source could be primarily traced to used third-party libraries instead of their own source code for four of the five applications.

REFERENCES


**AUTHORS’ BIOGRAPHIES**

**Sören Frey** is a PhD candidate at the University of Kiel, Germany. He received his MSc in computer science from the Mannheim University of Applied Sciences, Germany, in 2007. His research interests include software architectures, software reengineering, and cloud computing. Particularly, he is currently interested in model-based migration and the restructuring of software architectures towards efficiently employing multi-tenant application frameworks. Here, the focus lies on optimized resource utilization in platform as a service and infrastructure as a service environments. He is a member of the ACM, the IEEE Computer Society, and the German Association for Computer Science.

**Wilhelm Hasselbring** is a full professor of software engineering at the University of Kiel, Germany, and spokesperson of the Kiel/Lübeck KoSSE competence cluster on software systems engineering. Previously, he was a full professor of software engineering at the University of Oldenburg, chair of the TrustSoft Graduate School on Trustworthy Software Systems, and a scientific director in the OFFIS technology transfer institute. His research interests include software engineering and distributed systems. He received his PhD in computer science from the University of Dortmund, Germany, in 1994. He’s a member of the ACM, the IEEE Computer Society, and the German Association for Computer Science.
Benjamin Schnoor received his BSc in computer science from the University of Kiel, Germany, in 2010. He currently participates in a master’s program that has the focus on Software Engineering. His research interests include software architectures, cloud computing, and dynamic software analysis.