An Extensible Architecture for Detecting Violations of a Cloud Environment’s Constraints During Legacy Software System Migration

Sören Frey and Wilhelm Hasselbring

Software Engineering Group
University of Kiel
24118 Kiel, Germany
{sfr, wha}@informatik.uni-kiel.de

Abstract—By utilizing cloud infrastructures or platforms as services, SaaS providers can counter fluctuating loads through smoothly scaling up and down and therefore improve resource- and cost-efficiency, or transfer responsibility for the maintenance of complete underlying software stacks to a cloud provider, for instance. Our model-based approach CloudMIG aims at supporting SaaS providers to semi-automatically migrate legacy software systems to the cloud. Thereby, the analysis of conformance with the specific constraints imposed by a cloud environment candidate along with the detection of constraint violations constitutes an important early phase activity. We present an extensible architecture for describing cloud environments, their corresponding constraints, and appropriate violation detection mechanisms. There exist predefined constraint types with specified domain semantics as well as generic variants for modeling arbitrary constraints. A software system’s compliance can be examined with the assistance of so called constraint validators. They operate on discovered KDM-based models of a legacy system. 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.

Keywords—Cloud computing; Cloud environment constraints; Constraint validation; CloudMIG; Migration to the cloud; KDM

I. INTRODUCTION

Adopting cloud computing technologies 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 [1]. 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 [2] 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 simplify 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. Here, every constraint validator refers to a certain constraint type. For cases where the standard constraint validators might not yield sufficient results or 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. This 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 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 reconstructed the architectural models of five open source systems and investigated their intrinsic CEC violations by applying the validation process.

The remainder of the paper is structured as follows: Section II provides an overview of key aspects when migrating legacy software systems to the cloud with a focus on CECs. In Section III, we describe the extensible architecture for detecting CEC violations. Section IV presents the quantitative evaluation. The related work is described in Section V before Section VI draws the conclusions and outlines the future work.

II. MIGRATION TO THE CLOUD

A. 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 question of which data assets 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 [3].

Regarding technical challenges of a migration we identified the following major shortcomings of current approaches [2]: 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, automated 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 being briefly described in the next Section II-B 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).

B. The Approach CloudMIG

CloudMIG is our approach for supporting reengineers to semi-automatically migrate existing software systems to cloud-based applications [2]. 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:

A1 Extraction: Includes the extraction of architectural and utilization models of the legacy system. They base upon OMG’s Architecture-Driven Modernization1 efforts and utilize the Knowledge Discovery Meta-Model2 (KDM) and Software Metrics Meta-Model3 (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 transformation of the existing system from the generated target architecture to the aimed cloud environment.

C. Cloud Environment Constraints

1) Definition: To clarify concepts, we explain the constraint-related key terms below 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 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 appraisalise of a person modeling a cloud environment. Therefore, it should be rather seen as a hint for the reengineer. It is more important to detect a CEC violation at all and to make the reengineer beware of it.

2) Examples:

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.

3) Management with CloudMIG: CECs are made explicit in CloudMIG. Cloud profiles are modeled according to the CEM (see Section III-A) 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 are 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. 1 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 II-C2 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 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.

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.
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. Fig. 2 outlines our prototype tool CloudMIG Xpress and the integration of the 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. 2 exhibit their affiliation to CloudMIG’s activities through related activity numbers.

III. THE ARCHITECTURE FOR DETECTING CEC VIOLATIONS

A. 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.

1) Overview: The CEM is organized in layered packages as presented in Fig. 3. The Core package includes basic elements like abstract cloud services or partitions. The last allowing both Amazon EC2’s Availability Zones and Regions, for example. 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 OMG’s SMM. The Constraints package models the CECs and is covered in Section III-A3 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 Deltacloud4 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. 4. 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. 4 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 the concrete instances in other layers may not be sufficient. The alignment with KDM is described in the following Section.

2) Alignment with KDM: Besides domain-specific elements of the Cloud Profile package, one can recognize

the alignment of CEM with KDM in Fig. 4. Generally, CEM builds upon KDM’s platform and structure packages following the piggyback pattern [4] 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 inheriting from KDM classes (gray) in Fig. 4 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 lightweight 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 a KDM PlatformElement and therefore can not incorporate the EnvironmentConstraintConfiguration class (a PlatformElement as well, see Fig. 5) via a composition, as this association does not exist for two PlatformElements. It rather has to utilize an instance of KDM’s PlatformRelationship (the class CloudEnvironmentConfigurationContains). Listing 1 illustrates this concept by means of a KDM Google App Engine for Java cloud profile extract in XMI notation. For example, the PlatformModel in line 12 represents CEM’s CloudEnvironment, as its stereotype attribute refers to the according KDM Stereotype in line 2. The virtual meta-model element CloudEnvironment is in turn contained in a KDM ExtensionFamily\(^5\) describing CEM’s Cloud Profile package (line 1).

\(^5\)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”

---

**Listing 1.** Google App Engine for Java Cloud Profile (KDM excerpt).

```xml
1 <extension name="cloudprofile">
2  <stereotype name="CloudEnvironment" type="PlatformModel"/>
3    <tag tag="id" type="String"/>
4    <tag tag="providerName" type="String"/>
5    <tag tag="version" type="String"/>
6  </stereotype>
7  <stereotype name="CloudEnvironmentConfiguration" type="PlatformElement"/>
8    <tag tag="id" type="String"/>
9  </stereotype>
10 </extension>
11
12 <model xsi:type="platform:PlatformModel">
13    <stereotype name="CloudEnvironment"/>
14    <tag name="/extension.0" @ stereotype.0."/>  
15    <value "cloudmig.cloudprofiles.gae"/>
16  </platformElement>
17  <taggedValue xsi:type="kdm:TaggedValue"/>
18  <ownedRelation xsi:type="platform:PlatformRelationship" to="/model.0" from="/model.0"/>
19 </platformElement>
20 </model>
```

---

**Figure 5.** The “Constraints” package of CEM (excerpt).
Concrete CECs are modeled by inheriting from AbstractConstraint which provides an attribute for a ViolationSeverity, for instance. In Fig. 5, some concrete CEC examples are listed for illustrating purposes, 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 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).

IV. EVALUATION

To study characteristics of CEC violation detection 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. Our prototype tool CloudMIG Xpress was utilized to process the reconstructed models of these systems and to perform the validation process to detect CEC violations. An exemplary summary of a single validation run can be seen in Fig. 7. In the context of our evaluation, for each CEC violation detected a data record was reported comprising the IDs of the context of our evaluation, for each CEC violation detected a data record was reported comprising the IDs of the validation run can be seen in Fig. 7. In the context of our evaluation, for each CEC violation detected a data record was reported comprising the IDs of the validation process with a call to the method initialize and validate, respectively. In the case 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.

Concrete CECs are modeled by inheriting from AbstractConstraint which provides an attribute for a ViolationSeverity, for instance. In Fig. 5, some concrete CEC examples are listed for illustrating purposes, 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 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).
A. 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 III-A1 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 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. Moreover, 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 following open source applications were selected for evaluation purposes:

App1: Coefficient Core V. 0.9.6
App2: iBATIS JPetStore V. 4.0.5
App3: JavaBB V. 0.99
App4: jForum V. 2.1.9
App5: Ace Operator V. 1.7.0

Some basic characteristics of each system can be found in Table I. Considering LOC of 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.

B. Methodology

For extracting the related KDM models of the applications, we applied the tool MoDisco 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 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” representing method calls among others. 29 CECs were modeled, it was possible to detect 20 CECs that could be covered by our provided 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 II 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, it 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 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?

Additionally, for every question stated above we were interested whether we could identify indicators for patterns regarding a correlation with the systems’ sizes.

The number of classes and libraries for each program could be extracted from the KDM models. As a further sufficiently well-suited size measure we employed LOC. It was measured with CLOC V. 1.52 omitting comment and blank lines. However, it was only possible to measure LOC for the applications’ own sources, as the sources for the third-party libraries were missing. However, third-party libraries were only incorporated for analyzing Q6 and for that question LOC is not relevant.

C. Results

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

Q1: Fig. 8 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 size comes a steady growth in the nr. of classes raising CEC violations, but without a uniform growth rate or pattern.

Q2: Fig. 9 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.

Figure 10. Distribution of detected violation types (w/o third-party libs).

Figure 11. 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.

violations, too. But for CEC violations with Warning as violation severity this statement is weaker.

**Q3:** In Fig. 10 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 IV-B. 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. 11 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. 12 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:** In Fig. 13 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. 13 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.

### D. 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 II-C3 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 II 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.

Moreover, 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 Q6 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 web-based Java software systems. Only representatives of this application type were studied. The number of probands was also rather small.

Overall, generalizability is limited but the experiments provided us valuable insights and showed that our approach can be successfully utilized.

V. RELATED WORK

The reengineering of software systems builds a primary domain our approach is located in. An overview of legacy software system migration was contributed in [5]. 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 [6] propose several techniques to reduce migration complexity, for example dynamic program analysis, software visualization, and knowledge discovery.

The novel field of cloud computing constitutes a major foundation of our work as well. [7] provides an overview of 20 cloud definitions and extracts a consensus definition. The authors in [8] survey the research published in this area and discuss lessons learned from related technologies. A related case study was conducted in [9]. 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. [10] reports on a migration of a large scientific database to the cloud.

VI. CONCLUSION AND FUTURE WORK

We presented our extensible architecture for detecting a legacy software system’s violations of constraints imposed by cloud environments when considering a migration. It is an important constituent of our approach CloudMIG which supports reengineers in migrating existing software systems to the cloud. Here, the detection and highlighting of crucial system parts is an essential early phase activity. Violations may be easy to fix, but a reengineer has to be aware of them. Furthermore, our approach provides support for assessing the severity of a detected violation.

The future work includes further analyses of constraints impeding a migration to the cloud. Moreover, it is planned to model additional cloud profiles, provide a public repository, and to improve detection capabilities. For example, incorporating runtime information constitutes a promising approach. Furthermore, it will be interesting to examine application types that differ to a greater extent.

REFERENCES


