Rule-Based Architectural Compliance Checks for Enterprise Architecture Management

Constanze Deiters, Patrick Dohrmann, Sebastian Herold, and Andreas Rausch
Clausthal University of Technology
Department of Informatics – Software Systems Engineering
P.O. Box 1253
38670 Clausthal-Zellerfeld, Germany
{constanze.deiters, patrick.dohrmann, sebastian.herold, andreas.rausch}@tu-clausthal.de

Abstract

Modern enterprise application systems are parts of complex IT landscapes. The architecture of such a landscape may impose constraints upon the design of single applications, for example by the mandatory use of enterprise-wide reference architectures. It is of great importance for the sake of smooth operation and easy maintaining that single applications are compliant to the reference architectures. Checking this compliance is highly important for the architecture management to assure the quality of application systems. Unfortunately, current tool support is not flexible enough to easily check different aspects of architectural compliance.

This paper proposes a rule-based approach based upon logic programming concepts towards a formalism for architectural compliance checking. In this approach, the architecture and design are represented as logical knowledge base that can be queried for architectural compliance. Furthermore, the paper presents a case study, in which the approach was prototypically implemented and applied in an industrial context.

1. Motivation

Nowadays, there is still no agreement on a single precise definition of the term “software architecture”. A bunch of definitions can be found at [1]. Most definitions share a common view of the notion “software architecture”, which is more or less similar to the definition in the corresponding ANSI/IEEE Standard [2]: Software architecture is defined by the recommended practice as the “fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution”. Thereby, one essential goal of software architecture is to provide a structure of components and their interrelationships to meet the behavioral, non-functional, quality, and life-cycle requirements of the considered application [3,4,5].

Based on the level of abstraction there exists a clear distinction between the notion of architecture, design, and implementation, as discussed and formalized in [6] and [7]. However, as already included in the ANSI/IEEE definition software architecture “contains the principles governing design and evolution” of the corresponding application. Hence, to attain one of the most important goals of software architecture, we have to guarantee that the elaborated and possibly even verified architecture is correctly further refined and transformed into design and implementation of the corresponding application.

Moreover, modern enterprise applications are part of complex IT landscapes. Enterprise Architecture Management (EAM) claims to provide an approach to manage and evolve these IT landscapes by an IT management process as strategic business demands and by providing standards and guidelines for enterprise architectures for the development of single enterprise applications [8]. Enterprise architectures of IT landscapes define reference architectures for all enterprise applications. Thereby, reference architectures structure and specify interfaces between and within the enterprise applications to document and minimize the dependencies between the corresponding enterprise applications.

Similar to software architectures, designs, and implementations of single applications, it is also a crucial task of the involved software architects and developers to ensure the compliance of these artifacts towards the more abstract, coarse-grained, and application-spanning reference architectures. For example, the
design of a single application, like an information system, might be restricted by the enterprise-wide reference architecture defining the separation into mandatory logical layers [9]. If this structure is violated by not allowed dependencies, the separation into different concerns that motivated the layers gets obsolete and may affect the whole IT landscape. As a consequence, it might become impossible to maintain and to adapt. Thus, compliance checking helps to ensure the quality of the enterprise application and the overall IT landscape it is embedded in.

Even highly sophisticated experts, like the Eclipse developer team, are not able to completely fulfill their own architectural rules, like simple import rules between packages [10]. Obviously, for large enterprise applications and IT landscapes this architectural compliance check cannot be manually performed due to size and complexity of the related applications, but must be supported by proper tool support [11]. Unfortunately, current tool support, like for instance SonarJ [12] or Architecture Rules [13], is not flexible enough to easily check different aspects of architectural compliance, but focus on single aspects like checking architectural layer structures in code.

Instead of checking the compliance between the architecture and the design or implementation one could argue using Model-Driven Development (MDD) [14] approaches would solve the problem. Thereby, models on different levels of abstraction and (semi-)automatic transformations between them can be used to refine an abstract model, like the architectural model, to a more specific model, like the design model, and finally generate parts of the implementing code. If the transformations preserve the architectural semantics the final design and implementation is compliant with the intended architecture.

This approach does not really solve the problem: For instance, one abstraction the architectural model provides is that it is an intensional model [6]. To generate the concrete design model elements out of the architectural model, it would have to be an extensional model and, thus, would not provide a higher abstraction level. Furthermore, since a transformation into a refining design model cannot be complete, we would have to check the architectural compliance after manual changes anyway.

For that reasons we claim an approach to support architectural compliance checks between the intended architecture or reference architecture and the design and implementation of enterprise applications. The architectural compliance check is part of an overall enterprise architecture management approach.

The required tool support for architectural compliance checks must consist out of the following basic building blocks (see fig. 1):

- A formal notation to represent the intended architecture or reference architecture.
- A formal notation to represent the design or implementation of the application.
- A formal notation to represent mappings between intended architecture and design/implementation.
- A compliance checking technique based on the representations of architecture, design, and mapping.

This paper tackles these issues and introduces a rule-based approach to compliance checking. It was applied and evaluated in an industrial case study, in which the approach was used to check whether the implementation of a real-life enterprise application was compliant to a given reference architecture.

The remaining paper is structured as follows. Section 2 introduces the foundations of the proposed approach. The industrial case study and the related experiences are described in section 3. Section 4 gives an overview of related work. Section 5 concludes the paper.

2. The approach at a glance

This section introduces the proposed approach to rule-based architectural compliance checking. It bases on formal representations of architecture and design descriptions as logical knowledge bases. After an overview of the overall approach and its foundations, the formal representations will be described. The section ends with a description of the compliance checking technique and its prototypic implementation.
2.1 Overview and foundations

Common sense in the understanding of an architectural description is its task to describe the architectural building blocks and their connections and the principles that guide the further design and evolution of the system (see sec. 1). These principles can be seen as constraints for further descriptions that are only valid refinements if they obey the constraints of the more abstract description.

The constraints, that such a description defines, normally have free variables which are assigned to concrete objects of a semantic domain for which the constraints have to hold. Usually, that domain is infinite, and so are the possible valid assignments. Eden and Kazman [6,7] call such descriptions intensional and show that architectural descriptions (they use the term architectural specification instead) are intensional. For example, consider an architectural description that describes a system that is architecturally divided into two layers \texttt{l}\textsubscript{lower} and \texttt{l}\textsubscript{upper} (see fig. 1). It might constrain a Java implementation in that way that classes in packages representing \texttt{l}\textsubscript{upper} (p\textsubscript{1}, p\textsubscript{2}) might import classes from packages representing \texttt{l}\textsubscript{lower} (p\textsubscript{3}) but not vice versa. Figure 1 shows a graphical design description for Java that violates this constraint because \texttt{C}\textsubscript{2} imports \texttt{C}\textsubscript{6}. Nevertheless, there are infinite many Java programs that obey this constraint, thus the architectural description is intensional. On the other hand, we call a description with a finite number of possible implementations extensional. In the following, we call the constraints, which an architectural description defines, architectural rules.

Thus, checking architectural compliance means to show whether a design or implementation description (e.g. the code of a system or a UML model of it) fulfills the architectural rules of the architectural description. In our approach, we use first-order logic to represent the architecture and design description elements and the architectural rules as well. Figure 1 illustrates the proposed approach. It bases on a description triple – the architecture and design descriptions whose compliance has to be checked and, in addition, a mapping description. The latter relates elements from the architecture description to elements of the design description, e.g. packages that are assigned to layers. It has to be created by the software architect before the compliance check can be executed. Every of the three descriptions is represented as a single logical knowledge base which contains facts (and potentially logical rules) representing the single elements of the descriptions. Additionally, the architectural knowledge base comprises rules that reflect the intensional character of the architecture description. It contains rules with free variables that can be instantiated with objects from the union of the knowledge bases. If all of these rules are satisfied for a given triple of descriptions as described above, meaning that the rules hold on the union of the knowledge bases, the design description is said to be compliant to the architecture description.

2.2 Formal representation of architecture descriptions

As mentioned in section 2.1 the architecture description is represented as logical knowledge base consisting of logical facts and logical rules. While the facts represent (in an extensional manner) the existing elements in that description, the architectural rules define constraints upon the triple of descriptions depicted in figure 1. For the sake of brevity, the available predicates are restricted to those that are necessary for the case study in section 3. Therefore, we will consider layers [9] and allowed dependencies relations, which define that dependencies may only exist between elements of certain layers. Layers are identified by unique identifiers. Instances of those predicates are depicted in the lower left part of figure 1.

The architectural rules that are imposed by an architectural description upon a design description are expressed as logical rules that have to hold upon the union of knowledge bases of architecture, mapping, and design descriptions. The predicates we can use to formulate these rules depend therefore partially upon the kind of design descriptions. We defer a detailed description of the rule definitions to section 3, after the representation of design descriptions is introduced.

2.3 Formal representation of design descriptions

To transform design models into logical knowledge bases, the available predicates have to be known. For example, consider UML models as design descriptions [15]. The information that the UML element “Class” is transformed into a unary predicate “Class(\texttt{?x})” (\texttt{?x} being a placeholder for an identifier) has to be known before a specific class “Book” can be transformed into a corresponding fact “Class(2)”, whereas “2” identifies the class “Book” (see figure 3). As one can see, that information about the available predicates can be derived from the meta model.

The left part of figure 2 shows a small cutout of the UML meta model that depicts how “Classes” can be connected by “Associations”, “Class” is a specialization of “Classifier”, “Type”, and “NamedElement”
from whose it inherits meta attributes or meta associations like “name” from “NamedElement”. The connection between classes and associations is defined in the meta model by “Property”. On the one hand, a property is used to model attributes of classes. On the other hand it can represent an association end. In this case, it is owned by an instance of “Class” and is referred to by an instance of “Association”, or just the other way around. These alternatives are modeled by the two meta associations between “Property” and “Association”.

From this meta model cutout logical predicates and rules can be derived systematically. The relevant elements to capture structural properties of an instance of this meta model, a specific UML model, are the meta classes and their properties, namely meta attributes and meta associations. For each meta class, we can define a unary predicate, e.g. Association/1, which is true, if and only if the argument is an instance of that class. Furthermore, the generalization structures between meta classes are reflected by logical rules of the form ClassGen(?x) :- ClassSpec(?x) indicating that if ?x is an instance of “ClassSpec” it is also an instance of “ClassGen”.

Meta attributes are represented as binary predicates. A predicate <class>_<attr>(?c,?value) is true, if and only if the instance “c” of “Class” has its attribute “attr” allocated with the value “?value”. Meta associations are represented by defining binary predicates for their ends. <class>_<assocEnd>(?c,?d) is true, if and only if the instance c of “Class” is connected with an instance d (of some other meta class) by an association so that d plays the role indicated by “assocEnd”.

The right part of figure 2 shows the according predicates and rules for the UML meta model cutout depicted in the left part. There are seven unary predicates for the meta classes of the cutout. The rules reflect the specialization relationships. For the meta attribute “name” of “NamedElement” a predicate namedElement_name/2 exists, for the meta association end likewise, e.g. class_ownedAttribute/2 or prop-erty_class/2.

Figure 3 shows in the left part a small UML model in its concrete syntax as class diagram and aside as an object diagram showing its abstract syntax. Two classes “Author” and “Book” are connected by an association “wrote”. Simplified, this leads to five instances of meta classes: two classes, one association and two properties representing the association ends.

The predicates, which represent the meta model of UML, can now be used to create facts for this specific model (see right part of figure 3). It is assumed that we can uniquely identify the model elements. For every instance of a meta class a corresponding fact, like Class(1) for the class “Book”, is created. Accordingly, their instances of meta attributes are represented, like for example namedElement_name(1,”Book”). Please notice, that it is possible by the defined rules to identify the class “Book” as an instance of “NamedElement”.

The connection between the instances is represented by facts according to meta association ends. For example, the ends of the association (id 4 and 5 in figure 3) are connected with the association (id 3) by the facts association_memberEnd(3,4) and associa-tion_memberEnd(3,5).

2.4 Formal representation of mapping descriptions

Assuming, that the architecture descriptions look like introduced in section 2.2 and that design descriptions are models written in UML (see sec. 2.3), we are able to define some very simple mapping description that determine, e.g how layers are mapped to UML elements of the design description. Packages are mapped to layers – if the package p is interrelated to a layer l by a mapping, elements that are contained in p or in one of its direct or indirect subpackages are also assigned to l. Since layers have unique identifiers (see
sec. 2.2) and UML packages have a unique qualified name (see [15]), both are unambiguously identifiable. Thus, a mapping between a layer and a package can be defined by a binary predicate (exemplary depicted in fig. 1). This mapping is necessary to formulate the architectural rules, as one will see in section 3.

2.5 Compliance Checking Technique and prototypical implementation

The knowledge bases generated from the descriptions (see fig. 1) can now be used to check whether the architectural rules defined by the architectural description hold. The union of the knowledge bases can be interpreted as a program written in a logical programming language like Prolog. The checking of architectural compliance is thus technically realized as executing the architectural rules as queries upon the union of knowledge bases. For user convenience, we classify architectural rules into prerequisite rules and constraint rules. Prerequisite rules hold if their execution as query is successful, this means there is at least one result. Constraint rules hold if their execution as query returns no result. The distinction has practical reasons. By the use of constraint rules we are not only able to detect whether an architectural rule does not hold but also to detect the elements that cause a rule violation, since the corresponding query results would give information about the violation. Examples of both kinds of rules will be illustrated in section 3.

The approach was prototypically implemented as Eclipse plugin in [16]. The prototype was developed to generally query structural properties of a UML model that is represented as knowledge base. It is able to generate facts from a given UML model, for instance a design model, and to load user defined facts and rules, for example derived from an architecture description. After loading and generating all necessary information, the user can execute predefined queries upon the knowledge base or create new queries himself and execute them – for example queries consisting of the architectural rules of an architecture description. As query language and engine, a prolog-like programming language system named TyRuBa is used (introduced in [17]). Figure 4 shows a screenshot of the query dialog where the user can execute arbitrary queries. The depicted query, for example, determines all classes with their names from the loaded model. After execution, the result of the query is presented in a separate view (see lower part of figure 4).

In addition to the query engine the prototype includes two subsystems: a meta model transformer and a model transformer. The meta model transformer generates predicate definitions for TyRuBa out of an arbitrary meta model which is loaded as an Ecore file by the EMF framework [18]. The model transformer generates facts for a given UML model which is available as XMI file. To generate the facts for the UML model, the transformer uses the predicate definitions that were generated by applying the meta model transformer to the UML meta model. During the model transformation the whole model is traversed and facts for model elements (instances of meta classes) and instances of meta attributes and meta associations are created. The values of the corresponding predicate parameters are read from the model.
3. Case study

This section first describes the environment in which the presented approach and prototype were applied to evaluate the underlying concepts introduced in section 2.1 to 2.4. As example of use the architectural rule concerning the layered architecture is formulated in terms of logical facts and rules and executed against the design description. This design description is UML-based and is conform to the Java-based implementation. Finally we sum up our results and discuss possible reasons for the identified architectural violations.

3.1 Setting and architecture

Our experiences applying the above described approach are taken from an industrial cooperation project as part of a process for quality assurance. This part was intended to check the implementation of a medium-sized information system developed by an extern service provider against the agreed logical reference architecture. However, a satisfying concept and tool support had not existed so far.

The considered system is, according to the layers pattern, subdivided into different layers (see fig. 5). Like a usual information system, it consists of a part containing the application logic (layer application core), a part handling the data access (layer persistence) and a part providing functionality for user interaction (presentation and client) or interaction with external applications (web services). In figure 5 layers are depicted as boxes and allowed dependency relations as arrows, while used layers are located at the arrowheads and using layers at the ends.

According to the mentioned layer architecture several architectural rules the considered system has to obey can be identified. The first rule follows straight from the layered architecture: if layers and dependencies between these layers are defined, the design description and the corresponding program code have to fulfill these constraints. At this point we can directly attach a second rule: Before checking for compliance with the layered architecture there has to be assigned components/packages to each layer. Following the notation introduced in section 2.5 the first rule is a constraint rule and the second one a prerequisite rule.

3.2 Required architecture and mapping facts

The queries corresponding to the before mentioned architectural rules are formulated using the information in the knowledge bases regarding existing layers, assigned packages, and allowed dependencies between these layers. This information is represented by facts using the predicates layer/1, allowedDependency/2 and assignPkgToLayerByQualifiedName/2, whereas the first two belong to the architecture description and the third to the mapping description. E.g., the fact layer("applicationcore") indicates that applicationcore is a layer’s name. Furthermore, allowedDependency("presentation","applicationcore") expresses that dependencies from presentation to applicationcore are allowed.

To fill the layers with “life”, e.g. assignPkgToLayerByQualifiedName("org::anonymous::administration","applicationcore") assigns package org.anonymous.administration from the design description to layer applicationcore. Such packages which are directly assigned to a layer are called layer packages here. In contrast to the architecture and mapping description the design description bases on the UML meta model which allows to automatically generate appropriate predicates and facts using the prototypic tool introduced in section 2.5.

3.3 Defining the logical rules

To formulate the aforementioned architectural rules in terms of logic programming some logical auxiliary rules are helpful which will be defined in the following. Every dependency connects two elements of the design description, whereas each element belongs to a layer package. These elements and the corresponding layers are required to identify whether a dependency violates the layered architecture or not. Elements are arranged in packages which in turn are elements, too. To examine the layer which an element belongs to, the containing package is iteratively identified until a layer package is reached. The easiest case is the one in which the element itself is a layer package. In listing 1 this fact is expressed with the first elementInLayer/2. The second elementInLayer/2 describes the other case, in which the considered element lies somewhere inside the hierarchy beneath the layer package. First, the element’s surrounding package (ele-
Listing 1. Definition of logical rules to assign packages to layers and check whether an element belongs to a layer

```
packagesIsLayer(?pkg, ?layer) :- Package(?pkg), layer(?layer), packageByQualifiedName(?pkg, ?pkgName), assignPkgToLayerByQualifiedName(?pkgName, ?layer).

elementInLayer(?el, ?layer) :- Package(?el), packagesIsLayer(?el, ?layer).
elementInLayer(?el, ?layer) :- element_owner(?el, ?nestingPkg), isSubpackageOrTheSame(?nestingPkg, ?layerPkg), packageIsLayer(?layerPkg, ?layer).
```

Listing 2. Definition of logical rules to check, if dependencies between two elements exist

```
elementDependsOn/2 has to be identified and then the package hierarchy is navigated upward until a layer package was found (isSubpackageOrTheSame/2, packageIsLayer/2). The logical rule defined by packageIsLayer/2 evaluates whether a package corresponds to a layer. Used as a query with fixed layer package this rule returns the related layer; but only if an existing logical fact has assigned this package to a layer before.

Between two elements various kinds of dependencies might be exist:

- **Directed relations.** Thinking about directed relations for descriptions corresponding to Java code mainly generalizations, interface implementations, and package imports have to be considered.
- **Associations.** Associations are often specified with a direction of navigation between its assigned classes or interfaces.
- **Attributes.** Attributes of classes can be typed by other elements of the design description, like classes or interface.
- **Operation parameters.** Operations may have call or return parameters whose types refer to other elements of the design description.

To examine the dependencies used in the design description a logical rule was formulated for each above mentioned case. The first alternative of elementDependsOn/2 in Listing 2 covers directed relationships. The element ?fromEl depends on ?toEl if a directed dependency (EXISTS ?dep) with source ?fromEl and target ?toEl exists. After finding a result logical resolution tries to find more results by using backtracking. Therefore, the expression EXISTS ?dep forces the evaluation to stop after finding one dependency.

Associations and classes are connected with each other by association ends. An association end is owned by either a class or by the association itself. These two cases regarding the ownership have to be distinguished and mapped on adequate logical rules as they are depicted in Listing 2 at the second and third position. The second alternative of elementDependsOn/2 describes the case that the ownership of the association end belongs to the association. In reference to the UML meta model [15] association ends are instances of the meta element property (Property/1). Now, there is a dependency from ?fromEnd to ?toEl if they are classifiers (Classifier/1), like classes or interfaces, and at least one association (Association(?dep)) between them exists. The choice of relevant association ends (?fromEnd and ?toEnd) is limited to following constraints: ?toEnd has to be navigable and owned by the association (association_navigableOwnedEnd/2), and the property ?fromEnd is owned by the association ?dep (property_association/2). Finally, the type of both association ends is identified (typedElement_type/2).

If a class holds the ownership of an association end (third alternative of elementDependsOn/2), the element ?fromEl has to be a class which owns the association end ?toEl (property_class/2). Finally, ?toEnd has the type of classifier ?toEl which leads to the depending of ?toEl on ?fromEl. Attributes are also instances of the UML meta model element property. Since they are therefore modeled similar to association ends, the third alternative of elementDependsOn/2 also covers dependencies due to attribute types.

Call or return parameters in operations may also cause dependencies. To detect such dependencies from classifier ?fromEl to ?toEl the fourth alternative of elementDependsOn/2 in Listing 2 was formulated. If an operation ?op exists which belongs to ?fromEl (feature_featuretingClassifier/2), and further,
For example, the query then a dependency from exists. The predicate describing all variables free in the above defined logical (auxiliary) predicates. Leaving description. They can be noted formally by using section represents the intensional part of the architecture, which are visualized in figure 6. The darker arrows identify the violations and the numbers at each end of the arrows inform about the quantity of the layer’s involved classes or interfaces.

Additional, we were able to identify the precise classes and investigated the reasons for these violations in more detail: A lot of violations between the presentation layer and the client result from using objects for the transportation of data between these layers by data transfer objects [19]. These objects were defined inside the client and were used by the presentation layer. Maybe it is a better solution to move the definitions to the presentation layer. The violations between application core and client appear due to a kind of searching functionality that was implemented in the client layer. The utility layer contains security functionality which accesses the persistence layer and therefore produces violations. Some violations are caused by imports which are used to realize callbacks, like needed for the observer pattern. The latter was found between application core and web services.

3.5 Lessons learned and future work

The case study and the requirements of the participating partners show that architectural compliance checks in the field of enterprise application systems are highly necessary. The partners confirmed the usefulness of the approach because of the rapid feedback produced by the prototype. The results of the checks were very helpful to identify conceptual design errors or errors made in haste that the participating architects were very helpful to identify conceptual design errors produced by the prototype. The results of the checks were very helpful to identify conceptual design errors or errors made in haste that the participating architects were not even aware of. Executed in regular intervals, those checks would have been able to prevent those errors. It also showed that the formalism was efficient,
at least for the size and complexity of the system under investigation. A complete check whether the design description was compliant to the layers of the logical architecture took about 90 seconds on a common desktop PC. The creation of a corresponding knowledge base took about 5 minutes. Thus, the complete process could easily be integrated into an automatic, regularly executed process like a nightly build of the project. Of course, more detailed performance analysis, especially for larger systems have to be made. Eventually, this may lead to more efficient but less expressive formalisms like description logics [20], which are less computationally complex.

Furthermore, our partners found the declarative way of defining architectural rules, after some initial familiarization, easy to understand. Anyway, the approach has to be further elaborated w.r.t. architectural rule definitions because it is an error-prone task yet. Small details in the rule bodies can lead to execution times higher by magnitudes. A first version of the third alternative of \texttt{elementDependsOn/2} (see listing 2), which additionally tested for the existence of a connecting association instance leads to execution times of hours. Further work will focus on a meta model for descriptions for which the underlying knowledge base will be generated automatically.

Additionally, the approach is very flexible w.r.t. the kind of rules and documents that have to be checked. It was also used to formulate rules that check whether the design description (which reflected the Java code structure) was properly structured into components. Beside design descriptions, many different artifacts exists in practice, in which architectural rules should be enforced, for example, configuration documents like XML files for frameworks like Spring [21] or Hibernate [22]. For example, it might be necessary to define that Hibernate mapping files may only refer to classes in the persistence layer. Facts of those artifacts can be easily added to the knowledge base by defining corresponding predicates and facts. [16] describes how predicates can be automatically generated, independent of any specific meta model. However, in the case of adaption for other descriptions and artifacts, it might be necessary to adapt the architectural rules. If these changes are too complex, it would be a drawback to adaptability.

4. Related Work

In section 1, the work of Eden and Kazman [6,7] was already mentioned, which bases on similar thoughts as the approach proposed by this paper. However, their approach bases on the assumption that there is a common semantic foundation of all specification languages, namely design models [7]. Whether these are always adequate in practice for arbitrary description techniques and artifacts must be further elaborated.

Related work regarding architectural compliance checking can be found in the field of consistency management for models, an overview for UML models can be found in [23]. There exist numerous approaches to consistency management and checking consistency between arbitrary models. For example, [24] proposes an approach based on Communicating Sequential Processes (CSP), which might be more appropriate for behavioral consistency aspects. Other approaches are based on graphs and graph transformations, as for example [25] and [26]. However, examples mostly do not deal with the consistency or views on the same or a similar level of abstraction, and not with consistency towards an intensional model.

There are also approaches that deal specifically with the consistency of architecture and design models. For example, [27] deals with checking architectural features in component diagrams by using description logics. Contrarily, [28] uses the same logics to detect design inconsistencies. Egyed, for example, outlines an approach in [29] for consistency checking between C2 ADL [30] architecture descriptions and refining UML high-level designs. For this approach holds the same limitation as for other Architecture Description Languages [31]: they do not provide description techniques with intensional character.

On the implementation level, there are similar approaches applying logic programming concepts. [32] describes the application of Prolog to query arbitrary models. It is not explicitly applied to architecture models of enterprise applications, but has similar foundations as our approach.

5. Conclusion

The experiences from the case study presented here, the results, and the lessons learned show that architectural compliance checking is an important but difficult task for EAM and an important part for the assurance of the quality of enterprise applications.

A rule-based approach can provide enough flexibility to tackle this task. A (software) architecture does not only consist of layers but has many aspects that are relevant w.r.t. compliance checks. For that reason, the user (the software architect) should be able to define his own, customized rules. This is possible with the proposed approach.
The case-study also shows that the use of logic programming concepts is an appropriate founding formalism. However, its potential will be fully exploited when it is possible to define architectural rules by models like UML models that are common to software architects and to hide the logic-based formalism.

To further develop the approach, we will investigate how a meta model for architecture descriptions should look like that truly reflect the intensional characteristics of software architectures. Furthermore, the application to larger and more complex case studies will be useful to investigate whether the approach scales well or whether a change towards less expressive logics like description logic might be reasonable. Last but not least, we will identify more useful architectural rules in the field of enterprise applications to support controlling and managing their architecture and, thus, to help assuring their quality.

References