2013 17th IEEE International Enterprise Distributed Object Computing Conference

Checking Conformance with Reference Architectures: A Case Study

Sebastian Herold, Matthias Mair, Andreas Rausch, Ingrid Schindler
Department of Informatics - Software Systems Engineering Group
Clausthal University of Technology
P.O. Box 1253, 38670 Clausthal-Zellerfeld, Germany
Email: firstname.secondname@tu-clausthal.de

***Abstract*—Reference architecture can help in enterprise architecture management to develop and operate standardized and maintainable software landscapes. Similar to the software architectures of single systems, however, they are threatened by architecture erosion, i.e. the continuous divergence between intended architectures and their actual realizations. Architecture erosion has negative effects on the maintainability of software systems and on other quality attributes.**

In this paper, we report on the application of a rule-based architecture conformance checking approach in an industrial case study in which we investigate an industrial reference architecture for the German public administration. The reference architecture and its constraints for implementations are formalized as architecture rules enabling automatic conformance checking tool support.

The results from the case study show that the approach is capable of checking reference architecture conformance in realistic settings and helps to avoid software architecture erosion.

***Keywords*-reference architectures; software architecture erosion; architecture conformance;**

I. Introduction

The software architecture of a software system describes its most fundamental structures embodied in components, their interfaces, and the relationships between components as well as principles and guidelines for the further evolution of the system [1]. It manifests the earliest and most far-reaching design decisions and influences significantly the ability of a software system to meet its functional and extra-functional requirements [2].

Enterprise architecture management, however, is often faced with complex software landscapes in which the elements—the single software systems—have been developed by different providers according to different software architectures. This increases the complexity of the software landscape and decreases understandability and maintainability. Reference architectures can help to keep the complexity of software landscapes controllable; they define best practices, architectural principles, architecture patterns, and guidelines for a class of related software systems, for example, software systems for a shared application domain [3]. Simplified, reference architectures can be understood as software architectures for families of software systems.

Software architectures, however, are only useful if they are refined and implemented correctly and if this achieved architectural conformance is preserved. The opposite effect, the divergence between intended architecture and its refinement and realization, is called architecture erosion [4] and leads to negative effects on quality attributes like adaptability or maintainability [5]. In the long run, continuously and uncontrolled eroded software systems become too expensive to be maintained anymore and need to be replaced by expensive re-developments [6].

According to [7], there are three strategies to deal with architecture erosion. The first one subsumes approaches that aim at completely avoiding architecture erosion. These are largely approaches that couple architecture design and implementation tightly by integrating architecture concepts into single programming languages (e.g., ArchJava [8]). This is in general hardly possible in practical, heterogeneous settings. The two other strategies, minimizing and reversing erosion, subsume very different approaches, for example, consistency management in MDE [9], reflexion modeling [10], or other dependency analysis approaches (e.g., [11]). They all have in common that they require a way to detect architectural erosion, for example, via conformance or consistency checking. The result are violations of the intended architecture or architecture violations for short. Due to the size and complexity of modern software systems, this detection cannot be done by hand but needs tool support.

Especially for reference architectures, the applied detection mechanism must be very flexible. A reference architecture subsumes many different practices, guidelines, and patterns that define the constraints that a realization has to adhere to in order to conform with it. Common tools, for example, SonarGraph [12] or Structure 101 [13], focusing on single architectural aspects or patterns like the logical layering [14] of a system, are not powerful enough for this task.

In this paper, we present the application of a general architecture conformance checking approach in an industrial case study. This approach continuously applied over the software life cycle can detect and avoid architecture erosion. In the case study, we formalized a reference architecture for software systems of the German public administration in order to be able to check software systems for conformance with that reference architecture. We applied the approach and a prototypical implementation on real-life systems and

1541-7719/13 $26.00 © 2013 IEEE
DOI 10.1109/EDOC.2013.17

present exemplary results.

The remaining paper is structured as follows. Section II introduces the investigated reference architecture—the Register Factory®—and the architectural aspects that will be checked for conformance. Section III describes the applied approach to architecture conformance checking: it requires (a) specifying the reference architecture as a meta model such that the architecture of a system can be modeled and (b) the formalization of architectural aspects needing to be checked for conformance as logical expressions. Section IV describes in detail how this approach is applied for the Register Factory®and an exemplary system. The results of the conformance check and their reasons will be discussed. Section V will discuss related work; Section VI will conclude the article.

II. The Register Factory® Reference Architecture

Reference architectures define best practices, architectural principles, architecture patterns, and guidelines for a class of related software systems and can be understood as software architectures for families of software systems.

The Register Factory® is a reference architecture that was developed by the Capgemini Holding GmbH for the German Federal Office of Administration. It provides a well-documented standard for the creation of software in the environment of governmental institutions. The objective of the Register Factory® was the creation of a foundation for the design, realization and description of software systems for public administration [15]. Registers that are a fundamental part of public administration, like the commercial register or the central register for motor traffic, are used to manage structured data.

A. Structure of the Register Factory®

The Register Factory® is composed like a construction kit and contains the following constituent parts [15]:

- Building blocks: Building blocks are reusable components of software including functional and technical services.
- Operating platform: An operating platform is appropriated that defines a homogeneous infrastructure and its constituent parts. This enables a standardized system operation.
- Methodology: For every phase of software development the methodology of the Register Factory® offers several tools, for example, specification or modeling guidelines.
- Tools: Since the Register Factory® uses automation and tool support for the creation of software systems, it contains pre-configured tools, for example, for programming or error tracking.
- Blue-line Prints: *"Blue-line prints describe the architecture and concepts of software systems as well as their embedding into the application landscape from three different views: the functional view, the view of the software technical realization and the technical infrastructure view."* [15]

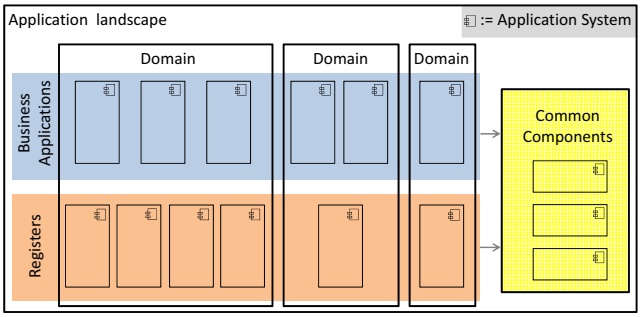

Figure 1. Functional view of an application landscape [16].

In this paper we focus on the blue-line prints, because this part includes some patterns that are used in the conformance check and described in detail in the following sections.

Figure 1 shows a simplified version of the functional view of an application landscape. It prescribes how application systems can be integrated into an application landscape. It can mainly be divided into Domain and Application system, for example, Registers, Business Applications, and Common Components. Furthermore, an application landscape contains a portal for web access and a service gateway for external usage. Registers basically manage and provide data for other components. Business applications use the provided data of registers as foundation for the description of abstract functionality. Furthermore, registers provide common components support and several services for logging, authentication, and commonly used functionality.

The technical view describes how the elements of the functional view can be mapped to software technical elements like components or classes [16]. As shown in Fig. 2 [15] the Register Factory® prescribes a modified 3-layer-architecture with a Persistence layer that manages the data access, an Application core and an Usage layer that consists of the parts: GUI, Batch and Service. This modified 3-layer-architecture is supplemented by a Common layer that supports severals services equivalent to the functional view.

B. Investigated Patterns

Section II-A outlined that the blue-line prints describe the architecture and concepts of a software realization which characterizes the structure of an application system. Such structures can be interpreted as patterns which define the basic organization and interaction between components. Due to page limitations, only five of the investigated patterns found in the Register Factory® are described in this paper. In the following an overview of these patterns is given.

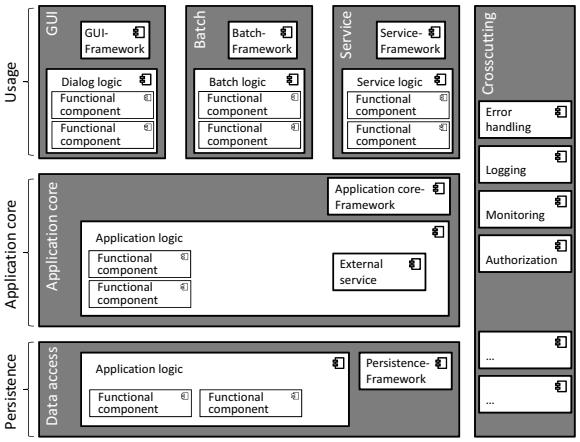

Figure 2. Technical view of the software realization of an application system [15].

1) *Component-Facade*: All communication across components has to be easy maintainable. Hence, each component should provide a specific interface for external usage, which restricts the access to defined methods.
2) *Exception-Facade for Component Interfaces*: It should be possible to access a component's interface in a defined behavior without interrupting the system. Therefore, a component interface is not allowed to throw any unhandled exceptions across the component border. This can be achieved by an exception facade implementation, which has to handle all method forwards to the original component interface.
3) *Exception-Facade Security*: An exception facade should not throw any undefined exceptions over the component border. To achieve this, every method call should be surrounded by a try-catch clause catching all possible exceptions and handle them by de-escalation or by throwing a method signature exception.
4) *Access Restriction to Component Facades*: A component should be easily replaceable and maintainable without effecting other components. The solution is to forward each component access via the component facade.
5) *Data Sovereignty*: All data have to be persistent in a controlled and consistent way by allowing only the component to access to its own data.

These patterns will be formalized in Sec. IV.

III. The ArCh Approach to Architecture Conformance Checking

In this section, the applied approach to architecture conformance checking is introduced which is named the *ArCh approach* after its implementing *Architecture Checker (ArCh)* prototype. The ArCh approach is very flexible with regard to checkable architectural aspects as well as supported

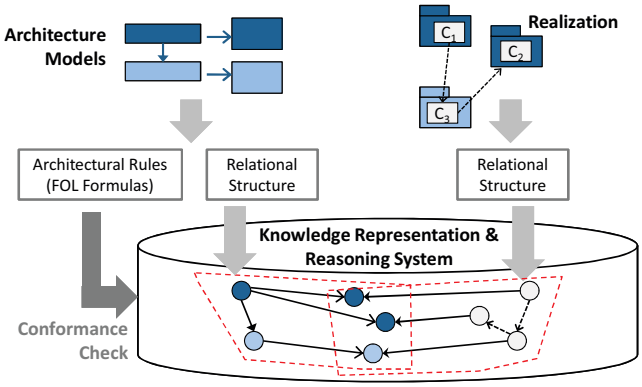

Figure 3. Conceptual overview of the proposed approach.

document types. The following subsections describe the core concepts and the implementation as prototype.

A. Concepts

The core of the ArCh approach is the formalization of architectural rules as first-order logic (FOL) formulas describing which properties a conforming system must have. Figure 3 illustrates the approach graphically. In the following, we will introduce the conceptual approach very briefly and focus on the realized tool support. More details on the conceptual aspects can be found in [17].

In this approach, models are basically represented as relational structures consisting of a universe of entities and relations between them [18]. A *signature* defines a set of relational symbols describing common object-oriented and architectural concepts; it determines which relations are "allowed", for example, to relate components and interfaces by a *provide* relation. Together with a predefined set of first-order logic axioms, the set of relational symbols defines an ontology for the representation of object-oriented and architectural structures. This ontology is easily extensible in contrast to a fixed meta model; if required, the set of relation symbols can be extended, for example, to express further architectural concepts, such as layers of the Register Factory® reference architecture (see Sec. II).

To represent models in this form, meta model specific transformation definitions have to be specified that express how model elements are represented as elements of relational structures according to the defined signature. For example, consider Java source code consisting of an interface I that defines a method signature *foo* returning an integer value. A formal representation[1] of this model could be a finite structure with the universe $U = \{I, foo, Integer\}$ and the relations $Interface = \{(I)\}$, $Signature = \{(foo)\}$, $PrimitiveType = \{(Integer)\}$, $definesSignature = \{(I, foo)\}$, and $hasType = \{(foo, Integer)\}$.

[1]The mentioned relations of the example correspond to relation symbols of the same name that are predefined in the signature.

Architectural models define in addition model-specific architectural rules which are expressed as first-order logic formulas specifying architecture-specific constraints. These formulas are instances of more general formulas defined for the architectural meta model by binding at least one of the parameters. For example, a formula $hasFacade(x)$ expresses that for the parameter x, which is considered to be a component, an adequate facade interface must exist according to the Register Factory® (see Sec. II for details). If we have the architecture of a specific system with the components *admin* and *analysis*, the formulas $hasFacade(admin)$ and $hasFacade(analysis)$ are generated for the specific model.

In this approach, architecture conformance formally means that the union of the relational structures representing the set of the system's models does not satisfy the formulas representing architectural rules defined by its architectural model(s). This is checked by executing the architectural rules as queries to a knowledge representation and reasoning system which contains the merged set of relational structures as factbase. In Sec. IV, we will describe in detail how instances of the Register Factory® can be transformed into relational structures and architectural rules.

B. Implementation

To evaluate the approach to architectural conformance checking, a prototypical tool, ArCh, has been implemented. It is based upon a realization using logic knowledge representation systems like Prolog or PowerLoom [19]. The overall structure and functionality is depicted in Fig. 4. The tool is integrated into Eclipse.

The main component of ArCh is the *Architecture Conformance Checking Framework*. It realizes the functionality of conformance checking as described in the previous section. It defines furthermore an internal representation of relational structures as used in the proposed approach. This is used by the wrapper interface to define a standardized interface to structures conforming to the common signature. Another interface, the backend interface encapsulates the specific knowledge representation system and keeps the implementation interchangeable.

Different wrappers encapsulate instances of different meta models (or programming languages) and provide a view as relational structures onto models (or source code) that need to be checked. Currently, wrappers exist for UML and Java as well as for several examples of architecture description languages such as a domain-specific language for Register Factory® architectures. For arbitrary meta models following the MOF [20], plugins generated by the Eclipse Modeling Framework (EMF) can be used to access models [21]. The transformation for Java has been realized as programmatic transformation using the Eclipse development tools that provide an API to the abstract syntax tree of Java source code. In addition, the Java wrapper has been

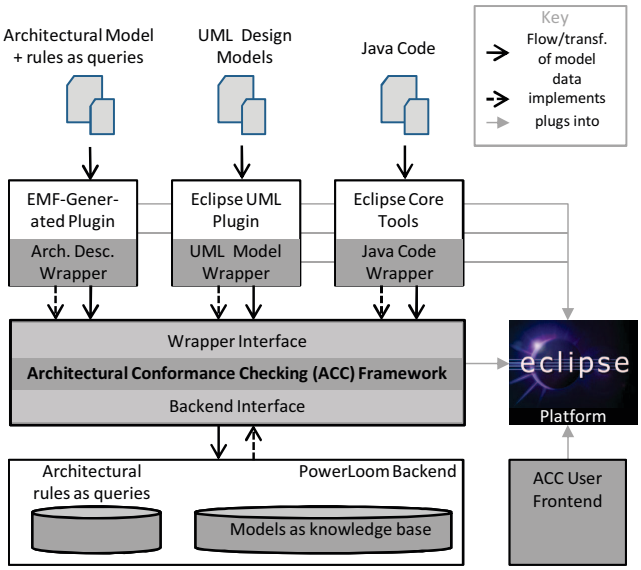

Figure 4. Prototypical tool concept for architectural conformance checking.

extended to deal with Spring XML files that are often used for declarative dependency injection (as, for example, in the systems considered in the case study). Most other available wrappers for ArCh deal with MOF-conforming meta models and implement programmatic transformations of models through using the EMF-generated API to navigate model structures.

The queries representing the logical sentences for the architectural rules that a meta model element implies are stored separately and serve as input for the corresponding wrapper. The framework also supports loading queries from libraries in which common queries can be stored for reuse.

The framework forwards the relational structures representing the models as merged structure to the connected backend which represents it specifically for the implementation as logical knowledge base; in the existing prototype, the PowerLoom system is used. Architectural rules are represented as logical queries and executed by the knowledge representation system in the process of conformance checking.

In this implementation, the relational structures are transformed automatically into code for a PowerLoom factbase. For example, if we assume again two components in the architecture *admin* and *analysis*, the following code is generated and stored into PowerLoom:

```
(assert (COMPONENT id_0))
(assert (COMPONENT id_1))
(assert (name id_0 "admin"))
(assert (name id_1 "analysis"))
```

These statements instruct PowerLoom to "assert" the described facts.

The transformation specifications for architectural rules

are stored in a separate XML file describing which logical queries have to be executed for instances of arbitrary meta model elements. For example, checking whether components have appropriate facade interfaces is enabled by the following XML cut-out:

```
<ruleCollection name="RegisterFactory Rules">
    <rule name="Component Facade">
        <code>hasFacade(?this)</code>
        <type>Component</type>
    </rule>
    ...
```

The parameter *?this* is always bound to the currently traversed instance of the meta model element when the model is transformed. The rule code describes which query should be executed for checking; the definitions of these queries are described in detail in Sec. IV.

IV. Conformance Checking Case Study

This section first describes the environment in which the presented approach and prototype were applied to evaluate the underlying concepts introduced in Sec. III. As example of use the architectural rules concerning the Register Factory® reference architecture are formalized as queries and executed by ArCh against the realization and their logical facts. The realization is a Java-based implementation of the Register Factory®. Finally, we sum up our results and discuss possible reasons for the identified architectural violations.

A. Environment and Setting

Our experiences applying the above described approach are taken from an industrial cooperation project. We investigated a Register Factory® based software system in the German Federal Office of Administration. Both the intended architecture and the realization of the software system have been provided by the industry partner. This made it possible to apply the approach against a reference architecture and its realization. The aim of the project was to clarify: Is the approach flexible enough to formalize the architecture rules of the Register Factory® reference architecture? Are there architecture violations in the realization?

Before executing the conformance check via ArCh, two tasks remain to be done. The first task was to formalize the reference architecture to describe Register Factory® instances. Thus, a meta model of the Register Factory® was created. The second task was to formalize the architecture rules as well. More specific, the following subsections describe the Register Factory® meta model and the formalization of architecture rules.

B. Register Factory® Meta Model

The intended architecture model of the Register Factory® is an instance of a meta model that describes the Register Factory® reference architecture. Therefore, we created an

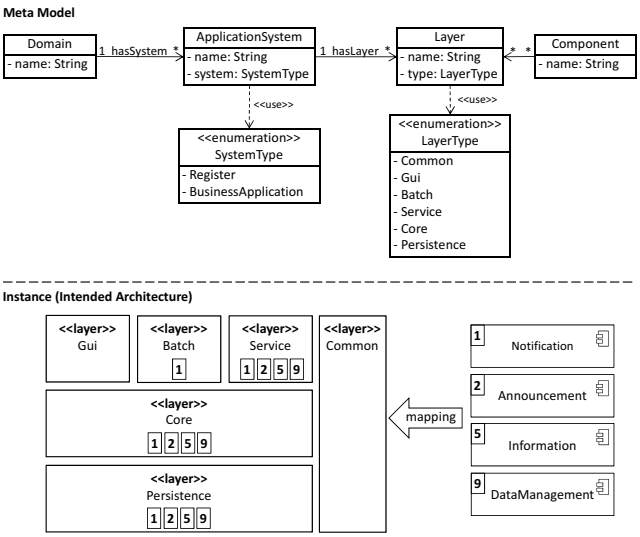

Figure 5. Register Factory® meta model and the intended architecture.

appropriate meta model which is depicted in the upper part of Fig. 5. The aim of the meta model structure is to describe a valid Register Factory® instance by modeling the component mapping to layers as outlined in Sec. II-A. More specific, the meta model describes the technical view of the Register Factory® structure as depicted in Fig. 2. A Domain has one or more ApplicationSystem that are specified via the enumeration SystemType and hence either a Register or BusinessApplication. An application system in turn has one or more Layer. Examples of layers are: Core, Persistence and Service. Finally, the meta model structure describes the mapping between Component and layers, a component is dedicated to an arbitrary number of layers.

After creating the meta model a new Register Factory® wrapper for ArCh was implemented. The wrapper's purpose is to transform an intended architecture—modeled by a software architect—to the relational structures and architecture rules. Therefore, the relation symbols are enhanced by the equal named relations in the meta model, for example, hasSystem and hasLayer. Within the transformation the logical facts of the enhanced structures are created and the components, in addition, are bound as parameter to the formalized architecture rules.

The provided intended architecture of the industry partner is depicted in the bottom part of Fig. 5. The focus of the intended architecture illustration is the mapping between components and layers and hence the upper hierarchical structures like domain or application system are hidden. Within the case study the four components Announcement, Notification, Information and DataManagement are checked for their architectural conformance in the realization. The mapping is represented in the illustration through component identifiers and their reuse as containment

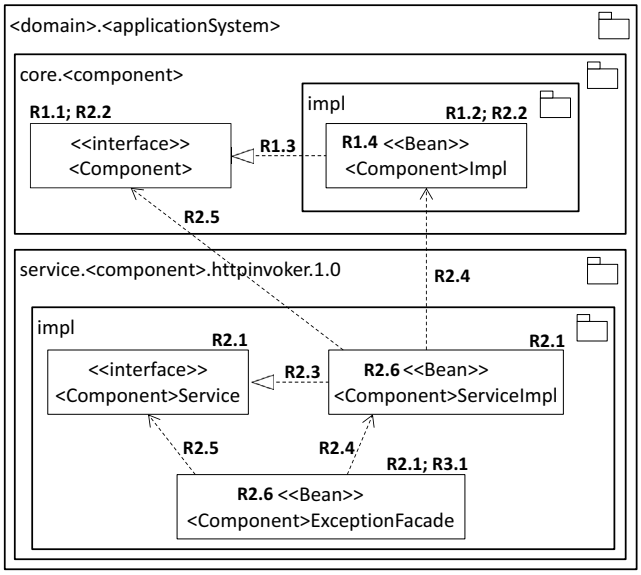

Figure 6. Component structure in the realization and the mapping to predicates via identifiers.

elements in the layers. For example, the announcement component with identifier 2 is mapped to the Service, Core and Persistence layer.

C. Formalization of Rules

The informal described architecture rules in Sec. II-B are formalized into the PowerLoom language, which provides a syntax for first-order predicate calculus expressions. Hence, the predicate calculus constructs complex sentences out of simpler ones using logical connectives. For specifying the architectural rules in PowerLoom the connective *and* and the declaration *defrule <name> (=> (precondition)(postcondition))* were used. Thereby, the declaration specifies in PowerLoom how to produce new knowledge based on the current knowledge. For example, a predicate in the precondition can check if a class name starts with an upper letter and the result and hence the new knowledge for an evaluated class A can be "does start with upper letter" or "does not start with upper letter".

The following subsections describe in detail the formalized architectural rules and their used predicates. For an easier understanding and orientation Fig. 6 illustrates a component structure in the realization and the mapping to predicates in the formalized rule via identifiers. For example, predicate identifiers are: R1.2 and R3.2.

1) Component-Facade: The "Component-Facade" pattern describes that each component containing to the core layer must provide an interface for external usage. Listing 1 depicts the PowerLoom implementation of this rule. The declaration `defrule` in line `1` asserts that, if the combined results of the predicates in the precondition (line `3`–`7`) are true, an evaluated component, represented by the variable *?comp*, is conform with the component facade (see line `9`). Hence, a negative result indicates an architecture violation.

```
1  defrule Component-Facade
2  (=>
3    (and
4      (isComponentInterface ?comp  ?if)      [R1.1]
5      (isComponentImpl      ?comp  ?clazz)   [R1.2]
6      (implements           ?clazz ?if)      [R1.3]
7      (existsSpringBean     ?comp  ?clazz)   [R1.4]
8    )
9    (hasFacade ?comp)
10 )
```

Listing 1. "Component-Facade" rule.

The PowerLoom syntax uses the prefix notation, thus, the logical connective `and` is followed by its arguments in line `4`–`7`. The predicate `isComponentInterface [R1.1]` checks whether an interface `?if` exists where the component name is equal the interface name and the interface is discarded in the package equal to the full qualified name of the component. For example, the "Announcement" component in the intended architecture (see Fig. 5) has the full qualified name *<domain>.<applicationSystem>.<layer>.Announcement* and according to this, the package name must be equal. The equal comparisons are case-insensitive. The predicate `isComponentImpl [R1.2]` checks if a class `?clazz` exists with the component name plus suffix "Impl" which in addition is located in the package equal the full qualified name plus ".impl". Thereafter, the predicate `implements [R1.3]` checks if the class from above implements the component interface. The last predicate `existsSpringBean [R1.4]` checks if a spring bean with the component name exists.

2) Exception-Facade for Component Interfaces: An "Exception-Facade for Component Interfaces" ensures that accessing the component interface should always result in the defined behavior. Therefore, it is not allowed that a component interface throws any unhandled exception across the component border. The solution is an exception facade which forwards all method invocations to the original component interface.

Listing 2 shows the formalized architecture rule. The first two predicates `isExceptionFacade [R2.1]` and `isComponentFacade [R2.2]` check the implemented component structure in the realization. Hereby, the first predicate checks if an interface with component name plus suffix "Service", a class with component name plus "ServiceImpl" and a class name plus "ExceptionFacade" exist. Furthermore, these types must be located in the sub package *httpinvoker.v1_0.impl* of the component service package. The second predicate checks if the component structure is similar to the exception facade. Afterwards, the predicate `implements [R2.3]` checks if the component service class implements the service interface.

```
1  defrule Exception-Facade
2  (=>
3    (and
4      (isExceptionFacade ?comp   ?sIf ?sClazz
5                                 ?sExcp)        [R2.1]
6      (isComponentFacade ?comp   ?cIf ?cClazz)  [R2.2]
7      (implements        ?sClazz ?sIf)          [R2.3]
8      (existsExSpringDep ?sClazz ?cClazz)       [R2.4]
9      (existsExSpringDep ?sExcp  ?sClazz)       [R2.4]
10     (existsUseDep      ?sClazz ?cIf)          [R2.5]
11     (existsUseDep      ?sExcp  ?sIf)          [R2.5]
12     (existsSpringBean  ?sExcp)                [R2.6]
13     (existsSpringBean  ?sClazz)               [R2.6]
14   )
15   (hasExceptionFacade ?comp)
16 )
```

Listing 2. "Exception-Facade for Component Interfaces" rule.

Next, the two predicates `existsExSpringDep [R2.4]` check if there exists an exclusive dependency, i.e. only declarative described Spring dependencies are allowed and thus no code dependencies between the specified parameters. Afterwards, the `existsUseDep [R2.5]` predicates check if a code dependency between the specified parameters exists. The last two predicates `existsSpringBean [R2.6]` check if the spring beans exist.

3) Exception-Facade Security: The "Exception-Facade Security" pattern ensures that an exception facade (see Sec. IV-C2) throws no undeclared exception over the component border by surrounding all method invocations with a Java *try-catch* programming clause. Thus, it is possible to catch all exceptions and handle them by de-escalation or by throwing a defined method signature exception.

```
1  defrule Exception-Facade-Security
2  (=>
3    (and
4      (isExceptionFacadeClass    ?comp  ?sExcp) [R3.1]
5      (implementsSignature       ?sExcp ?sig)
6      (throwsUndeclaredException ?sig   ?excp)
7    )
8    (hasExceptionFacadeSecurity ?comp ?sig)
9  )
```

Listing 3. "Exception-Facade Security" rule.

Listing 3 depicts the PowerLoom implementation of this architectural rule. The first predicate `isExceptionFacadeClass [R3.1]` checks for the existing of the exception facade class (see Sec. IV-C2). The second predicate `implementsSignature` ensures that only implemented signatures of the exception facade class are considered. Thereafter, the third predicate `throwsUndeclaredException` checks if all implemented signatures are compliant, i.e. no one is throwing an undeclared exception. These can be checked by a comparison of the defined signature exceptions and the throw statements in the signature implementation.

4) Access Restriction to Component Facades: The pattern "Access Restriction to Component Facades" ensures that each access to the component is forwarded via the component facade (see Sec. IV-C1) to keep the component easily replaceable and maintainable.

```
1  defrule Access-Restriction
2  (=>
3    (and
4      (isInternallyUsableOnly ?comp ?innerType)
5      (isNotPartOfComponent   ?comp ?type)
6      (existsUseDep           ?type ?innerType)
7    )
8    (trespassesCompFacade ?comp ?type ?innerType)
9  )
```

Listing 4. "Access Restriction to Component Facades" rule.

Listing 4 shows the formalized architectural rule. The first predicate `isInternallyUsableOnly` specifies the component's inner types, i.e. all component-related types without the component interface (see Sec. IV-C1) itself. The second predicate `isNotPartOfComponent` in turn specifies all the other types. Afterwards, the third predicate `existsUseDep` checks if there are code dependencies between outer types and inner types.

5) Data Sovereignty: The "Data Sovereignty" pattern ensures that the component data is persisted in a controlled and consistent way by allowing only the component to have access to its own data. Hence, all accesses from other components are architecture violations.

```
1  defrule Data-Sovereignty
2  (=>
3    (and
4      (isCompPersistenceOnly   ?comp ?persType)
5      (isNotPartOfCorePersComp ?comp ?type)
6      (existsUseDep            ?type ?persType)
7    )
8    (violatesDataSovereignty ?comp ?type ?persType)
9  )
```

Listing 5. "Data Sovereignty" rule.

Listing 5 depicts the formalized architectural rule. The predicates are similar to the rule "Access Restriction to Component Facades" (see Sec. IV-C4). The first predicate `isCompPersistenceOnly` specifies the component-related types in the persistence layer. The second predicate `isNotPartOfCorePersComp` specifies all types which are not component-related in the core or persistence layer. Thereafter, the third predicate `existsUseDep` checks if a code dependency between the component types in the persistence layer and the other types exists.

D. Results

The applied approach for conformance checking was executed against the Register Factory® instance (see Fig. 5), which includes four components and their Java-based realization. The implementation consists of about 900 classes

and interfaces, about 60,000 lines of code and 175 packages. From the realization about 250,000 logical facts were generated. The prototypical tool ArCh (see Sec. III) needed about 50 seconds[2] for generating the logical facts and about 6 seconds for conformance checking.

The result identified 8 violations in the realization regarding to the conformance of the intended architecture. Additionally, we were able to identify the precise components and classes and investigated the reasons for these violations in more detail: All violations are caused by the two components "Announcement" and "Notification" by violating the "Access Restriction to Component Facades" rule (see Sec. IV-C4) and the "Data Sovereignty" rule (see Sec. IV-C5).

After consulting the industry partner, the explained reason for 3 of 8 violations are allowed exclusions. One reason is that in the Register Factory® reference architecture specific components exist, in our example the component *BasicData*, which is allowed to violate the "Data Sovereignty". In addition, the input data classes of a component, i.e. all classes in a special sub package, are excluded in the "Access Restriction to Component Facades" rule. The remaining 5 identified architecture violations are confirmed by the industry partner and have to be dissolved in the realization.

E. Lessons Learned and Discussion

The measured time values in the above section showed that the performance of the approach is fast enough to be used in nightly builds or in continuously checks by developers and software architects. In contrast, the performance is too slow if just-in-time conformance checking is required that programmers know from checking for compile errors in most modern IDEs. They are used to get a quick response by the IDE if their source code changes (e.g, removing a variable, writing an expression, etc.) contain compile errors. The problem is based on the creation of the factbase, due to the complete recreation by modifications in the intended architecture or their realization. To increase the performance for just-in-time checking the prototypical tool ArCh has to be enhanced to support incremental changes, i.e. after an initial factbase creation all modifications are increments, which add or remove logical facts without recreating the factbase. Additionally, the used knowledge representation and reasoning system in ArCh has to support incremental changes as well to execute queries faster.

The result of the conformance check and their reasons showed that often exceptions are made which allow to violate these rules. Hence, it is possible that during life time a more common architecture rule gets more and more specific via excludes and other special behaviors. Due to that, we plan to use an easier understandable rule specification technique in the future, for instance, by using models or natural language.

[2]The time values are based on the execution on a common notebook with 4GB RAM and a Core i5 processor at 2.5 GHz.

Especially the last issue leads to a discussion of the pragmatics of a formally founded architecture conformance checking approach like the proposed one. The architectural rules constitute additional artifacts that need to be maintained along with the reference/software architecture of the system. Of course, applying and maintaining this additional mechanism for architectural conformance should pay off and should not eliminate the advantages of preserving architecture conformance by its costs.

It is not possible to determine a generally valid set of absolute numbers for project or system parameters at which the formalization pays off. However, it can surely be said that the easier the specification of architecture rules is the less effort must be put into maintaining and adapting them. Moreover, if rules can be reused the effort of creating and maintaining them will decrease. This can be the case, e.g., if there are predefined rules for common architectural concepts like patterns or of rules from existing systems can be reused in new development projects.

On the other hand, the costs of maintaining architectural rules for conformance checking must also be compared with the potential costs of repairing defects resulting from architectural violations. These are more expensive and more likely to happen, the larger and the more complex a software system is, and also the longer the system is developed and maintained, i.e. the longer it evolves. In case of reference architecture that are potentially implemented by a larger number of systems, the effort maintaining architectural rules might pay off even faster than for software architectures of single systems.

V. Related Work

There exist several related approaches to software architecture erosion detection and architecture conformance checking, respectively. For the special case of the Register Factory®, Brunnlieb's work [22] utilizes static code analysis to check the patterns of that reference architecture. The approach uses its own pattern description language and a prototype called Architecture Verification Engine (AVE) for verification. In contrast to the ArCh approach, which also allows checking of different artifact types easily, the prototypical realization of that work supports to check Java source code only for conformance. Moreover, in Brunnlieb's approach, the intended software architecture does not need to be explicitly modeled; instead, the existing components of a system are specified by a configuration file for the prototype. It is hence not easily extensible for arbitrary reference architectures.

In our own preliminary work, we developed an approach to conformance checking in enterprise architecture management [23]. That approach has in common with ArCh the idea to reflect the artifacts of a system in a knowledge representation system and to utilize a reasoning system. However, the preliminary prototype does not use a common

ontology like ArCh but instead can only check UML class/-component diagrams against a manually entered architecture rules. It was hence restricted to UML and furthermore to rules defined by the layers pattern.

The specification and formalization of patterns is a loosely related field of research [24]. These approaches focus on the specification, identification, and enforcement of mostly design patterns. The proposed approach aims more at the checking of arbitrary cases of architecture erosion than on a precise formalization of patterns which are only one source of architecture rules.

To our best knowledge, there is no published work dedicated software architecture erosion in reference architectures but of course the general approaches to architecture conformance checking are related research. Passos et. al [25] give an overview of several techniques as well as they illustrate the architecture conformance techniques at an information management system. One technique are reflexion models [10]. Intended architectures are modeled as modules and intended dependencies between these models. Modules are manually mapped to source code elements. Divergences between intended dependencies and existing source code dependencies are evaluated and visualized. Implementations of this technique are for example JITTAC [26], SAVE [27], ConQAT [28], or Bauhaus [29]. Architectural rules are reduced to dependency constraints and often only few artifact types, for example, meta models, are supported such that the flexibility regarding more complex reference architecture has yet to be proven.

Another technique is source code query languages. Query-language based approaches are often based on a relational calculus or on FOL. Some of these approaches are .QL [30], DCL [11], or CQL [31]; Kim Mens logic meta programming approach [32] belongs to this group of approaches and is probably the most relevant one compared with ArCh. However, most of these approaches do not easily integrate with arbitrary domain-specific modeling languages for architectures such as might be useful to describe the applied architecture concepts as in the Register Factory®. To our best knowledge, there are no public case studies of applying source code query language-based approaches to practically relevant reference architectures.

VI. Conclusion

As outlined in Sec. I, reference architectures can be understood as software architectures for systems of a common application domain. Reference architectures are hence an important and powerful concept for enterprise architecture management to define unified structures, concepts, and guidelines for application landscapes. The Register Factory®reference architecture described in this article, for example, is a very powerful mean for the systematic development of applications for the public administration.

Architectural erosion is a serious threat to the usefulness of reference architectures. If references architectures are not applied, refined, and implemented consequently, the overall quality of the software systems of the application landscape can suffer. In realistic and industrially relevant development scenarios with large and complex systems, heterogeneous technologies, and (globally) distributed development teams, software architecture erosion cannot be avoided completely. It is hence of great importance to be able to check architecture conformance regularly.

This article has described the ArCh approach to architecture conformance checking and has shown its application to the Register Factory®reference architecture. We have shown its applicability and ability to deal with practically relevant reference architectures by checking an industrial application system for architectural conformance. Architecture conformance is defined by architectural rules upon an ontology abstracting from specific document types/meta models. This concept makes the ArCh approach very flexible and hence well-suited for the purpose of checking reference architectures.

Besides being able to check the conformance with the intended reference architecture, the case study showed also positive effects on understanding the impacts and constraints of the reference architecture. The need to define a meta model for instances of the reference architecture and the need to formulate the architectural rules, improved the understanding of the reference architecture, its implications, pitfalls, corner cases, etc.

Nevertheless, the case study has also shown that there is still space for improvements. The definition of architectural rules should be more intuitive using modeling techniques that software architects from practice are more familiar with. The performance of ArCh is sufficient for batch-like conformance checking (e.g., part of nightly builds) or "once in a while" checking by developers and software architects; however, it would have to be improved if just-in-time checking is required. Finally, checking architecture conformance automatically is only the first step to more powerful tool support for dealing with software architecture erosion: analyzing the reasons for architectural violations and determining how to repair them are necessary subsequent tasks.

In our future work, we will investigate how to extend the ArCh approach with respect to these challenges such that it can support enterprise architecture management to deal with reference architectures and software architecture erosion even more efficiently.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley Longman, 2003.

[2] D. Garlan and M. Shaw, "An introduction to software architecture," Carnegie Mellon University, Tech. Rep. CMU-CS-94-166, Jan. 1994.

[3] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture - Foundations, Theory, and Practice*. Wiley, 2010.

[4] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40–52, 1992.

[5] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *J. Syst. Softw.*, vol. 61, no. 2, pp. 105–119, 2002.

[6] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam, "Modularization of a large-scale business application: A case study," *IEEE Softw.*, vol. 26, no. 2, pp. 28–35, 2009.

[7] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *J. Syst. Softw.*, vol. 85, no. 1, pp. 132–151, 2012.

[8] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation," in *Proc. of the 24th Int. Conf. on Software Engineering*. ACM, 2002, pp. 187–197.

[9] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Inf. Softw. Technol.*, vol. 51, no. 12, pp. 1631–1645, 2009.

[10] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: bridging the gap between design and implementation," *IEEE Trans. Softw. Eng.*, vol. 27, no. 4, pp. 364–380, 2001.

[11] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Softw. Pract. Exper.*, vol. 39, pp. 1073–1094, 2009.

[12] "Sonargraph website," accessed on 25th March 2013. [Online]. Available: http://www.hello2morrow.com/products/sonargraph

[13] "Structure 101 website," accessed on 25th March 2013. [Online]. Available: http://www.headwaysoftware.com/products/?code=Structure101

[14] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad, *A System of Patterns: Pattern-Oriented Software Architecture: Vol. 1*. John Wiley & Sons, 1996.

[15] S. Spielmann, F. Doerr, and F. Senn, "Register Factory: Eine Referenzarchitektur im Praxiseinsatz," *OBJEKTspektrum*, no. 3, 2012.

[16] Bundesverwaltungsamt, "Register Factory - Whitepaper," Tech. Rep., 2012. [Online]. Available: http://www.bva.bund.de/cln_321/nn_2148024/DE/Aufgaben/Abt__I/RegisterFactory/inhalt.html?__nnn=true

[17] S. Herold, "Architectural compliance in component-based systems. foundations, specification, and checking of architectural rules." Ph.D. dissertation, Clausthal University of Technology, 2011.

[18] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd ed. Cambridge University Press, 2004.

[19] "Powerloom website," accessed on 25th March 2013. [Online]. Available: http://www.isi.edu/isd/LOOM/PowerLoom/

[20] "Meta object facility (MOF) core specification version 2.0," Object Management Group (OMG), Tech. Rep., 2006.

[21] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF Eclipse Modling Framework*. Addison-Wesley, 2009.

[22] M. Brunnlieb, "Verification of Software Architectures using Static Code Analysis for Java," Master's thesis, University of Kaiserslautern, 2012.

[23] C. Deiters, P. Dohrmann, S. Herold, and A. Rausch, "Rule-based architectural compliance checks for enterprise architecture management," in *Enterprise Distributed Object Computing Conf., 2009. EDOC 2009*. IEEE, Sep 2009, pp. 183–192.

[24] T. Taibi, *Design Patterns Formalization Techniques*. Igi Global, 2007.

[25] L. T. Passos, R. Terra, M. T. Valente, R. Diniz, and N. C. Mendona, "Static architecture-conformance checking: An illustrative overview." *IEEE Softw.*, vol. 27, no. 5, pp. 82–89, 2010.

[26] J. Buckley, S. Mooney, J. Rosik, and N. Ali, "JITTAC: a just-in-time tool for architectural consistency," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1291–1294.

[27] J. Knodel, M. Lindvall, D. Muthig, and M. Naab, "Static evaluation of software architectures," in *Proc. of the 10th European Conf. on Software Maintenance and Reengineering (CSMR 2006)*, Mar. 2006, pp. 285–294.

[28] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens, "Flexible architecture conformance assessment with ConQAT," in *Proc. of the 32nd ACM/IEEE Intl. Conf. on Software Engineering (ICSE 2010)*. ACM, 2010, p. 247–250.

[29] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus — a tool suite for program analysis and reverse engineering," in *Ada-Europe*, ser. LNCS, vol. 4006. Springer, 2006, p. 71–82.

[30] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote address: .QL for source code analysis," in *Proc. of the Seventh IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*. IEEE Computer Society, 2007, p. 3–16.

[31] "Code Query Language Specification." [Online]. Available: http://www.ndepend.com/CQL.htm

[32] K. Mens, "Automating architectural conformance checking by means of logic meta programming," Ph.D. dissertation, Vrije Universiteit Brussel, 2000.