Complementing Model-Driven Development for the Detection of Software Architecture Erosion

Sebastian Herold and Andreas Rausch
Department of Informatics
Clausthal University of Technology
P.O. Box 1253, 38670 Clausthal-Zellerfeld, Germany
{sebastian.herold,andreas.rausch}@tu-clausthal.de

Abstract—Detecting software architecture erosion is an important task during the development and maintenance of software systems. Even in model-driven approaches in which consistency between artifacts can partially be established by construction and consistency issues have been intensively investigated, the intended architecture and its realization may diverge with negative effects on software quality.

In this article, we describe an approach to flexible architecture erosion detection for model-driven development approaches. Consistency constraints expressed by architectural aspects called architectural rules are specified as formulas on a common ontology, and models are mapped to instances of that ontology. A knowledge representation and reasoning system is then utilized to check whether these architectural rules are satisfied for a given set of models.

We describe three case studies in which this approach has been used to detect architecture erosion flexibly and argue that the negative effects of architecture erosion can be minimized effectively.

Index Terms—software architecture; software architecture erosion; architecture conformance checking; inter-model consistency

I. INTRODUCTION

The specification of an intended software architecture manifests the earliest and most far-reaching design decisions that are made during the development of a system [1]. It influences very strongly the quality attributes of software systems. Especially in complex or long-living systems, the phenomenon of architectural erosion or architectural drifts can be observed [2], i.e., the divergence of intended and realized software architectures. These effects threaten quality properties of the system like maintainability, adaptability, or reusability [3]. At long sight, uncontrolled architecture erosion leads to irreparable software systems [4].

Some definitions of software architecture pick up the issue of divergence and state in similar ways [1] that a software architecture must “contain the principles governing design and evolution.” In other words, the software architecture of a system provides a framework for the system’s detailed design and its implementation and restricts these further steps. For example, the layers of a logical architecture restrict the way relationships in UML design models can be designed, or how classes in Java source code may depend on each other.

Software architecture erosion is thus a consistency problem between artifacts of software development processes. Consistency between artifacts is also an important issue in model-driven software development (MDSD) which has been investigated intensively by research in the subfields of model transformations and inter-model consistency [5]. Nevertheless, Biehl et al. [6] were among the first who showed that architecture erosion also might occur in MDSD approaches.

One might assume that transformations ensure that high-level artifacts are consistently transformed into low-level artifacts. However, the transformations do in general not create the entire set of low-level artifacts. Instead, they create skeletons that need to be extended and completed manually. Due to this semi-automation, projects developed with MDSD are also prone to the problem of drifts between the various models on different levels of abstractions and views with different perspectives. Such inter-model drifts can be introduced for various reasons, such as manual additions, incomplete or incorrect transformations, and synchronization issues, as argued in [6].

Moreover, general purpose consistency checking approaches common in MDSD for checking consistency after transformation and manual manipulation of models are difficult to use for erosion detecting in typical development scenarios. In these approaches, the specification of consistency constraints depend on the syntax of the participating models, and hence had to be repeatedly defined for each kind of models in which architectural erosion might appear. This redundancy limits the usability of these approaches for detecting erosion.

The focus and contribution of this article is in the area of architectural design within MDSD approaches. We will present an approach which allows integrating effective architecture erosion detection into MDSD more easily than existing solutions. Different models are represented as instances of a common ontology and architectural consistency constraints, which are called architectural rules in the following, are expressed as logical formulas over these structures.

The remaining paper is structured as follows. Section II illustrates the inherent problems and complexity of architectural conformance/architecture erosion and summarizes existing potential solutions to these issues. In Sec. III, we propose a new approach for checking architecture erosion in MDSD and introduce its implementation as a prototypical tool. Section IV describes some case studies where the approach has been applied. Section V concludes the article.
II. Architectural Conformance in MDSD and Approaches to Detect Erosion

It is broadly accepted that there is a separation of architectural design and detailed design [7]. While architectural design is about architectural components and their relationships, detailed design is about forming the inner structure of those components [1]. Following this understanding, the design phase of software projects is often divided into three steps: architectural design, detailed design, and implementation whereas, depending on the specific process model, these steps are iteratively and repeatedly executed. In each of these steps artefacts are created — in MDSD: models — which provide views on the inner structures of a system at different levels of abstraction, adding more and more details starting at the most abstract view of the software architecture. Moreover, architectural design and detailed design define rules that restrict the content of refining artefacts.

This means that there is an inherent complexity in checking architectural rules and enabling powerful tool support:

1) The set of artefacts types—in MDSD this means the set of meta models used to describe the system—tht are possibly affected by architectural rules, is heterogeneous and differs from project to project. It is hence not enough to check source code written in a single programming language, more implementation and design artefacts have to be considered.

2) There is a large variety of architectural rules. Different architectural concepts, like patterns (e.g. layers [8]), reference architectures (3-Tier-Architecture [9]), or design principles (like Law of Demeter [10]) can define rules that need to be checked in design and implementation.

Tools for architectural conformance checking in MDSD must hence be easily adaptable to different meta models and have to support an expressive mechanism for the definition of architectural rules. As described briefly in Sec. I, model transformations as part of MDSD approaches are not sufficient to ensure architectural conformance. To illustrate this issue, consider the following simple example of a layered architecture [8]. The logical layers of a system should be transformed into packages of a UML design model to structure the detailed design model according to the layers. Let us assume an architectural meta model exists containing a concept “Layer”. The hierarchical structure that a specific layering introduces into the system implies constraints, i.e. architectural rules, that have to hold in the design — informally spoken, the constraint that upper layers may depend on lower layers but not vice versa.

Listing 1 shows by an ATL example [11] how a transformation rule for this case should conceptually look like. For a layer, a package of the same name is created to which a constraint is attached (by ownedRule) formulating the constraint enforcing the hierarchical dependency structure.

Listing 1. ATL rule to convert layers into UML packages with dependency constraint.

```java
module Arch2Design
create OUT : UML from IN : Arch
rule Layer2Package {
    from
    l : Arch!Layer
to
    p : UML!Package {
        name <- l.name
        ownedRule =
        'Nothing in this package depends on something contained in a package created as result of the transformation of a layer above l')
}
```

The conceptual problem with this solution is the dependency of the architectural constraint towards the structure of the transformation rule’s target meta model. The constraint as part of the target side of a transformation rule naturally refers to elements of the target meta model and is hence meta-model-specific. In realistic scenarios with different meta models in use for one software development project, a single architectural rule has to be manifested in many different sets of transformation rules. The constraint would have to be defined repeatedly for different meta models, causing overhead in maintaining the set of architectural rules. The same holds for approaches for inter-model consistency checking and model transformation like xlinkit, ECL/EVL, or similar approaches [12], [13], [5]. One approach to address architecture erosion in an MDSD setting directly is described in [14] which, however, focuses on the detection of patterns and seems less flexible w.r.t. heterogeneous settings with many different meta models needing to participate in conformance checking.

On the other hand, specialized approaches for detecting architectural erosion often do not integrate with MDSD very well, or have limited expressiveness regarding architectural rules. Dependency Structure Matrices (DSM) capture the dependencies between the modules of systems [15]. Lines and row represent modules of a system, entries indicate dependencies, either binary or weighted as numerical value. Tools like Lattix LDM allow software architects to define constraints like can-use to restrict the value certain entries are allowed to have [16]. Since architectural rules are more general than those dependency constraints, DSM provide only limited flexibility. Moreover, most approaches support only a limited set of design artefacts to be checked; furthermore, the integration of a high-level architecture description is not possible nor is such a description provided by tools themselves.

Reflexion models are another technique to check architectural conformance [17]. The architect creates an high-level model manually, containing the intended architectural modules and dependencies. A source code model is generated automat-

---

1 Please note that in the following consideration we will use the term “meta model” also for the definition of textual programming languages, i.e. for their grammar.
ially, reflecting the same dependencies between source code elements as they are. A manual mapping between architectural elements and source code elements allows it to depict differences in the set of relationships as so-called reflexion models. Implementations are for example SAVE, ConQAT, or Bauhaus [18], [19], [20]. Although they provide high-level models of the software architecture, and can hence potentially be better integrated into model-driven engineering, they reduce architectural rules to dependency constraints as well, and often support only few artefact types, i.e. meta models.

Most flexible with regard to different architectural rules are query-language based approaches like CQL, .QL, or JQuery which are query-language for object-oriented source code [21], [22], [23]. They often build upon a relational calculus or are based on first-order logics. Most tools, however, only support a small set of programming languages but not design models to be checked, and do not provide a high-level architecture model.

The research closest to the proposed approach is the logic meta programming approach proposed by Kim Mens [24]. He describes a framework to architectural conformance checking by representing architecture and implementation of a system by a logical knowledge base. A simple architecture description language allows to describe a system as a structure of concepts and relations between concepts. The predicates available represent Smalltalk source code by logical knowledge bases. The main difference compared to the proposed approach is that only architectural descriptions are understood as logical statements about implementation artefacts, causing the mapping to depend on implementation structure of object-oriented system as understood in Smalltalk.

III. PROPOSED APPROACH FOR DETECTING EROSION BASED ON A COMMON ONTOLOGY

It must hence be stated that the state of the art of detecting software architecture erosion in MDSD setting should be improved. Three main requirements concluded from the discussion above are:

- Support a broad set of detectable architectural rules.
- It should be possible to use one specification of an architectural rule to detect erosion in many different artefacts.
- Support the usage of models to define architectures and architectural rules as well as artefacts in which violation of architectural rules can be checked.

A. Overview of the Approach

The core of the proposed approach is the formalization of architectural rules as first-order logic formulas describing which properties a conforming system must have. Figure 1 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 [25].

In this approach, models are basically represented as relational structures consisting of a universe of entities and relations between them. A signature $\tau_{CBSD}$ defines a set of relational symbols and defines which relations are available; the relation symbols describe common concepts from component-based systems extending object-oriented concepts. Together with a predefined set $\Phi_{CBSD}$ of first-order logic axioms, the set of relational symbols defines an ontology for component-based systems. This ontology is easily extensible in contrast to a fixed meta model; if required, the set of relation symbols can be extended, e.g. to express domain-specific architectural concepts.

To represent models in this form, meta model specific transformation definitions have to be specified that express how model elements can be represented. For example, consider a UML class diagram stating that there is a component named Persistence which provides an interface Transaction enabling us to encapsulate database actions in transactions. A formal representation of this model could be a finite structure with the universe $U = \{\text{Persistence}, \text{Transaction}\}$ and the relations $\text{Component} = \{(\text{Persistence})\}$, $\text{Interface} = \{(\text{Transaction})\}$, and $\text{providesInterface} = \{(\text{Persistence}, \text{Transaction})\}$.

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) some of the parameters. For example, a formula $isValidLayer(x)$ expresses that for the parameter $x$, which is to considered a layer, no dependencies to upper layers might exists. If we have a specific architectural model with layers $l$ and $m$, the formulas $isValidLayer(l)$ and $isValidLayer(m)$ are generated for the specific model.

In this approach, software architecture erosion formally means that the union of finite 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 containing the merged set of relational structures as factbase.

In the example of Fig. 1, the architectural model consists of layers to which packages are assigned. In the design model (or graphical representation of the source code), packages are used to group classes. Architectural rules as above can now express in terms of the common ontology (enriched by the concepts of layers and allowed dependencies) what it means not to violate the given layering (no inheritance against the layering, no usage relationships, etc.). The merge of the relational structures matches the packages such that architectural rules can be correctly checked for conformance.

Note that following this approach, architectural rules do not depend on the syntactical structure of the models that need to be checked for architecture erosion but only on the structure defined in the common ontology and concepts optionally defined for the architectural meta model. Furthermore, architectural rules have the expressive power of first-order logic, and can be easily attached and checked on instances of arbitrary meta models as long as a transformation into the ontology can be defined. This definition has to be created once, when instances of a given meta model should be considered for architecture erosion detection, and can be reused, e.g. a transformation for Java and UML have been developed and reused in different projects. The creation of this transformation definition as well as maintaining this transformation and the architectural rules should be done by software architects that are aware of the constraints of their architectures; however, for common architectural principles, such as architectural patterns, the usage of reusable catalogs of architecture rules, would be desirable.

B. Prototypical Implementation

To evaluate the approach to architectural conformance checking, a prototypical tool has been implemented. It is based upon a realization using logic knowledge representation systems like Prolog or Powerloom [26]. The overall structure and functionality is depicted in Fig. 2. The tool concept is integrated into Eclipse.

The main component of the prototype 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 $\tau_{CBSD}$. 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 $\tau_{CBSD}$-view onto models (or source code) that need to be checked. Currently, UML and Java are supported as well as simple examples of architecture description languages. For arbitrary meta models following the MOF [27], plugins generated by EMF can be used to access models [28]. 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. At the moment, the transformation of MOF-conforming meta models is realized programmatically by navigating the model’s structure by the API provided through EMF. One of the next development steps, however, will be to replace the programmatic transformation by model transformations, e.g. in ATL, that create the internally used data model from EMF models.

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 the loading of 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; up till now, an implementation for Powerloom has been realized. 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 a Powerloom factbase. The example from the previous section with a component named Persistence providing an interface Transaction is translated into facts that are inserted into the factbase by assert statements:

```lisp
(assert (COMPONENT id_0))
(assert (INTERFACE id_1))
(assert (name id_0 "Persistence"))
(assert (name id_1 "Transaction"))
(assert (providesInterface id_0 id_1))
```

The transformation specification for architectural rules are
stored in a separate file, as described above. For example, the transformation detecting illegal dependencies between layers, respectively, is stored (accessible for the architectural meta model wrapper) as:

"Layer",
"(illegalDependenciesFromLayer
  ?this ?toLayer ?srcElem ?trgElem)",
"Actual layer dependencies are not compliant with intended ones!"

whereas the first entry refers to the meta model element to which the second entry, an architectural rule, applies; the variable ?this is replaced by the identifier of the layer that has to be transformed into the ontology. The third entry is a human-readable description of the rule. illegalDependenciesFromLayer is the corresponding predicate expressing the condition for dependencies that are forbidden by the actual layering:

(defrelation illegalDependenciesFromLayer
  ((?srcLayer LAYER)(?trgLayer LAYER)
   (?srcElem ELEMENT)(?trgElem ELEMENT)))

(defrule illegalDependenciesFromLayer_def
  (illegalDependenciesFromLayer ?l1 ?l2 ?e1 ?e2)
  (dependsOn ?l1 ?l2)
  (layerDependency ?l1 ?l2 ?e1 ?e2)
  (not (isAllowedToUse ?l1 ?l2))
  (not (= ?e1 ?e2))
  (not (exists(?s ?p)
    (INTERFACE ?i1)(INTERFACE ?i2)
    (exists(?s ?p)
      (SIGNATURE ?s)(PARAMETER ?p)
      (hasType ?p ?i2)
      (hasSignature ?i1 ?s)(hasParameter ?s ?p))))

First, this code declares a predicate for illegal dependencies from a layer srcLayer towards a layer trgLayer caused by the elements srcElem and trgElem. The second statement defines how to evaluate whether the predicate holds for a 4-tuple of entities. It basically says that no dependency between the elements may exists if the layers are not specified to be allowed depending on each other (“source” to “target” layer). layerDependency contains in its definition the further refinement of dependencies to inheritance relationships, usage relationships, etc. The following rule defines that dependsOn, to which the dependencies of a layer are refined, is true for two interface if one defines a method signature with a parameter typed by the other interface:

(defrule dependsOn_def4
  (dependsOn ?i1 ?i2)
  (exists(?s ?p)
    (SIGNATURE ?s)(PARAMETER ?p)
    (hasSignature ?i1 ?s)(hasParameter ?s ?p)
    (hasType ?p ?i2)))

The further refinement of layerDependency as well as the other rules for dependsOn are omitted for the sake of brevity.

IV. Case Studies

The proposed approach and its prototype have been evaluated in different case studies to show its flexibility in detecting software architecture erosion. The case studies differ at the architectural (meta) models that were used and the (meta) models requiring to be checked for erosion.

A. Information Systems with Layered Architecture

The first case study is a medium-sized information system of about 1,600 classes and 130,00 lines of third-party code and a given logical layer architecture that was determined together with the provider of the system. In [29], we applied a preliminary version of this approach not using a common ontology. Instead, it could only represent UML models formally and was hence only able to check a reverse-engineered UML model of the Java code. Moreover, there was no explicit architectural model but architectural facts had to be entered manually.

After this first experiment, we repeated the checks with the proposed approach and defined a simple meta modal for layers as required for the approach [25]. The rules were modified to expressions over the ontology for component-based systems expressing forbidden dependencies between layers and the contained elements. With our prototype and the existing UML and Java wrappers, we were able to detect the same points of erosion / architectural rule violations independently in the UML model of the system as well as in the Java implementation. Several hundreds of violations were detected that could be distinguished into five groups of different pairs of participating layers. Most of these violations could be reduced to conceptual problems like systematically wrong placement of classes with a dedicated functionality. This case study has shown that the implementation of the approach was able to handle medium-sized systems with an architecture at a level of detail common in industrial practice.

B. Architectural Patterns in Complex System

The second case study was conducted on the Common Component Modelling Example, an exemplary component-based system to compare component modelling approaches [30]. The domain of the system are retail stores like super markets. It handles sail processes at cashdesks as well as typical processes at the inventories of such stores like the check-in of delivered goods.

Although the system is relatively small with about 120 classes and less than 10,000 lines of Java code, it is interesting from the architectural point of view since it was constructed to integrate two different kinds of systems: a typical information system for the management of inventories and an embedded system controlling sensors at the cashdesk.

There are thus different architectural aspects that can be investigated for violation:

- Layered information system architecture enforcing a strict layering and separation between presentation, application logic, and persistence functionality.
- Service-oriented application layer in the sense that functionality of the application layer is provided by stateless
services delivering data as copies of persistent data (use of transfer objects).

- Embedded system with asynchronous communication between components through event channels only; components are not allowed to communicate directly but must send messages/events to a broker that forwards them to receivers.

In this case study, we decided to model the architecture and detailed design of the CoCoME as UML models with different profiles. The architectural profile contained the architectural concepts and attached rules for the mentioned aspects. A design model as well as the Java implementation were checked [31].

As a result we could state that the layering and the rules controlling the communication through events were not violated. The only architectural violation affected the component connecting the inventory with the cashdesk allowing the inventory to be updated immediately during purchases at the cashdesk. This component realizes event channel connection functionality as well as services to the inventory GUI and would possibly let GUI components access objects not transferred as required by the concept of a service-oriented layer.

The flexibility regarding different architectural aspects was a valuable property of the approach in this scenario.

C. Domain-specific reference architecture

In the most recent scenario, we aim at enabling architecture erosion detection for a domain-specific reference architecture at the German Federal Office of Administration (BVA). The BVA supports German ministries and their departments by providing administrative tasks centrally, such as payments of salaries, time and attendance recording, and many more. It recently introduced a reference architecture called Register Factory as a reference architecture for IT systems carrying out tasks around all different kind of registering in the public administration [32].

The reference architecture provide guidelines for technical platforms as well as the logical structure of application systems. Given a detailed but informal description of the main aspects of the Register Factory, we developed a domain-specific modelling language such that instances of the reference architecture—hence, architectures for dedicated application systems—could be modelled. It mainly defines the top-level structures of registers and application systems working on registers and proposes different layers for that systems in which application components must be structured.

Although the top-level structures and the corresponding rules seem similar to ordinary layering, the Register Factory makes detailed guidelines at a detailed level. For example, it defined precisely that components must provide certain facades for exception handling and service providing. We realized six different architectural aspects and formalized them as architectural rules. So far, we tested these rules on exemplary third-party systems with less than ten application components and about a few thousands lines of Java code with Spring dependency injections. However, evaluations on larger systems are planned for the immediate future.

This case study will be especially interesting regarding the consideration of different technical artifacts into the architecture erosion process (as already realized for Spring) and the size of the considered systems. Since we plan to accompany a new development project, it might be investigated, how far the approach can support the developers to avoid erosion by applying architectural conformance checks right from the beginning of development.

V. Conclusion

Tool support for architecture conformance checking in MDSD is important. Flexibility is needed with regard to the variety of architectural rules that restrict the refinement and implementation of an architecture as well as to the large number of different artifacts that are used in real-world development projects. In MDSD, this refers to different models and modeling languages for which it should be possible to use a single definition of an architectural rule (see Sec. III).

The proposed approach addresses these issues by using an expressive formalism in form of first-order logic and defining rules over a comprehensive ontology that abstracts from specific meta models. Although the mappings of meta model onto the ontology are still meta model specific, it is possible to specify and check architectural rules independently from the meta models of artifacts needing to be checked. In three different case studies, we applied the approach for different meta models and architectures successfully.

Future work will include evaluations in larger scenarios. This also includes formalizing a commonly used set of architectural concepts, e.g. popular patterns, as architectural rules, since the usability of this approach depends heavily on the reusability of existing rules. Furthermore, we work on approaches which start where pure conformance checking ends: in case architectural violations are found, they need to be repaired. At the moment, a recommendation system suggesting repair actions to developers and architects is developed.

The proposed approach allows to realize architecture conformance checking tools that are flexible with regard to supported meta models and to variability of architectural rules. An implementation framework allows such tools to be easily adapted to new meta models, such that existing conformance checking functionality can be easily enhanced to new models without the need to modify existing architectural rules.

This approach can complement existing powerful techniques in MDSD like model transformations which already help to tackle software architecture erosion partially by construction. It might hence efficiently help to minimize the negative impacts of architecture erosion on the overall quality properties of software.
REFERENCES


