The ArchMapper Approach to Architectural Conformance Checks: An Eclipse-based Tool for Style-oriented Architecture to Code Mappings

Simon Giesecke[1] and Michael Gottschalk[2] and Wilhelm Hasselbring[3]

[1] Simon Giesecke
BTC Business Technology Consulting AG
Kurfürstendamm 33
10719 Berlin
simon.giesecke@btc-ag.com
[2] Michael Gottschalk
freiheit.com technologies GmbH
Straßenbahnring 22
20251 Hamburg
michael.gottschalk@freiheit.com
[3] Wilhelm Hasselbring
Christian-Albrechts-Universität zu Kiel
Department of Computer Science
24098 Kiel
wha@informatik.uni-kiel.de

Abstract. The ArchMapper approach allows performing two activities in the software development process efficiently: checking the conformance of the code to the intended architecture as specified by an architectural description, and generating code skeletons and architecture-related configuration files from the architectural description. Both directions exploit information based on the architectural style of the software system. An architectural style may be as simple as the style of layered architectures, or it may correspond to a specific middleware platform, which allows more specific analyses and generation. We have applied the approach to the style of the Spring MVC framework, where several architectural properties can be checked, and the Spring configuration file for the application may be automatically generated from the architectural description.

1 Introduction

An important aspect in ensuring the evolvability of a software system is its conceptual integrity. Conceptual integrity is guaranteed on the architectural level by the adherence to a consistent architectural style. In practise, the adherence to an architectural style can only be ensured in the long term when tool support for checking the conformance of the implementation to the architecture and its style is available. This is particularly true in the context of distributed software development. Tool support requires that the architectural style can be formalised

in some way, which is not possible for all properties that can be associated with an architectural style. Strict approaches focus on declarative properties of an architectural style which require knowledge of a complex formalism in practically relevant cases (see, e.g., [1]). It is often easier, though less rigorous, to specify architectural rules by implementing checks for them imperatively. This does not require an evaluation engine for declaratively specified rules. The ArchMapper approach is such an approach. In this paper, we focus on using the ArchMapper approach to perform style-based architectural conformance checks as described before. However, the ArchMapper approach supports another activity, style-based code generation.

In the following, we begin by describing the conceptual foundations of the approach (Section 2). Then, we describe the architecture of the ArchMapper tool that implements the approach (Section 3). In Section 4, we describe an evaluation of the approach and the tool in a case study using a real-world system. Section 5 discusses related work, while Section 6 concludes the paper and provides thoughts on future work.

2 Concept

2.1 Foundations

Software Architecture and Architectural Views Currently, there is no consensus on the meaning of the term "software architecture" yet, so we briefly introduce our understanding. We distinguish the general term "software architecture" and subordinate "architectural views" (other authors and related approaches implicitly or explicitly equate "software architecture" with a specific "architectural view", e.g. [2]). This understanding is based on the definitions in the ISO 42010 Standard [3] for software architecture description, which defines software architecture as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution".

The architecture of a software system can be described from different viewpoints that may decompose the system into different kinds of elements, each of which is called an architectural view. The most important viewpoints [4] are the *module viewpoint*, which describes the structure of the code in terms of modules (in Java, e.g., these are usually identified with packages), and the *component-and-connector viewpoint* that describes the basic runtime structure of the system. When designing new software systems, it is convenient to use a straightforward mapping of components and connectors to modules, which means that certain modules are used exclusively by their corresponding components. However, library modules will typically be used by multiple components. When the software system is run on a runtime component platform, components may explicitly correspond to artefacts: For example, OSGi bundles (or Eclipse plug-ins) are components in this sense.

Architectural Styles Architectural styles can be defined for any architectural view, but they are most commonly used together with the component-and-connector view. Well-known examples are several variants of the pipe-and-filter style and of layered styles. For this architectural view, the following definition describes the seminal understanding of architectural styles: "An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined" [5]. An "instance" of a style is an architectural description of a concrete software system that conforms to the vocabulary and constraints of the style.

2.2 Style-oriented Architecture to Code Mappings

Our mapping approach is based on architectural descriptions in the component-and-connector view. The code however, is naturally organised in the module view. Therefore, the definition of the architecture-to-code-mapping implicitly involves a mapping between these architectural views.

Architectural information inherently goes beyond the information that is naturally contained in the source code. Any code-to-architecture conformance checking approach uses an architectural description and a mapping from the elements architectural description (e.g. components) to the elements of the code (e.g. source files, packages, and classes). A (generic) conformance checker uses the architectural description, the mapping and the source code to create a list of violations (if any exist). This basic approach is extended by the ArchMapper approach. An overview of its elements is shown in Figure 1.

In the ArchMapper approach, the architectural information consists of two types of artefacts: style descriptions, which are reusable for all software systems that are built on top of the same platform[4], and architectural descriptions, which correspond to a specific software system. In our evaluation, we use an academic Architectural Description Language (ADL) called Acme for describing styles and architectures. While this ADL is not well-known among software practitioners, it is easy to learn since it can be said to be a minimal language that has the required modelling features, i.e. the constructs necessary for modelling styles and instances of styles[5].

Acme has the additional benefit that a modelling tool is available, which is integrated with Eclipse and allows checking the architecture for conformance with the style, which means that these rules do not need to be checked directly in the code. However, there remain constraints imposed by a specific style that need to be checked in the code, e.g. constraints that refer to constituents

[4] An architectural style is a model of the platform from the component-and-connector architectural view. A platform is a specific mode of use of a set of middleware products. We distinguish platform and product since complex middleware products can be used in a variety of ways that incur architectural differences.

[5] We have shown that using the UML for this purpose is possible, but unfortunately awkward if implemented rigorously [6].

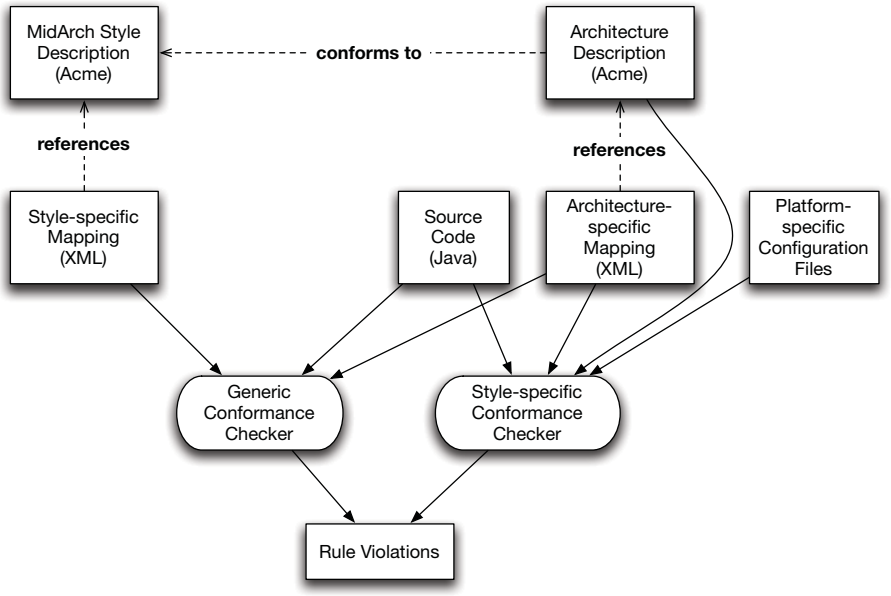

Fig. 1. ArchMapper Approach to Style-specific Architecture Conformance Checks

(e.g. methods) of the target elements of the mapping (e.g. classes). Therefore, the code-to-architecture conformance check can be improved by exploiting the knowledge that a software system should adhere to a specified architectural style.

This knowledge is used in two ways:

- A style-specific mapping. It describes rules for mapping style-dependent elements of an architectural description to the code. It is interpreted by the generic conformance checker that is used for style-independent mappings as well.
- A style-specific conformance checker. This may be used to check properties that cannot be easily expressed in the form of rules that the generic conformance checker interprets.

It is important to note that both the style-specific mapping and the style-specific conformance checker are specified or implemented independently from a specific software system. They can be reused and be applied to any system that is specified to adhere to the same style.

The properties checked by the generic conformance checker are the following:

Communicational integrity: For each dependency between elements in the code, an allowed dependency must be specified for the respective components in the architectural description.

Missing elements: Each component must be implemented by at least one code element, and every code element must be associated with a component.

3 Architecture

3.1 Foundations

Eclipse Static Analysis Tools The Eclipse Static Analysis Tools are part of the Eclipse Test and Performance Tools Platform. They provide a language-neutral framework and GUI for implementing and running static code analyses. A static analysis can be easily defined by implementing a Rule interface and defining an extension to an extension point supplied by the Static Analysis Tools.

3.2 Architecture of the ArchMapper Tool

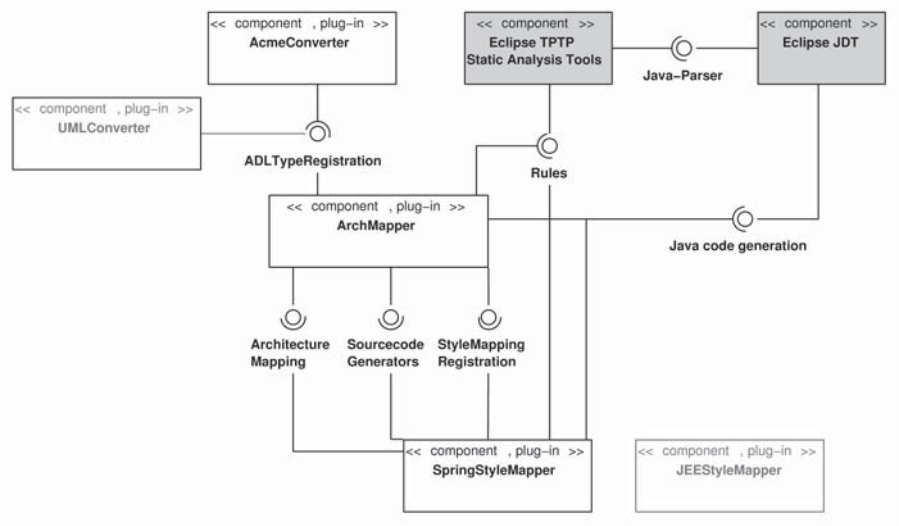

Fig. 2. Architecture of the ArchMapper Tool

Figure 2 shows an overview of the architecture of the ArchMapper tool[6]. For the sake of simplicity, the diagram shows dependencies for the case of Java. However, the architecture can be easily adapted to other languages supported by IDEs based on the Eclipse Workbench IDE, such as C++. In the current Java setup, the ArchMapper tool extends both the Eclipse Java Development Tools (JDT) and the Static Analysis Tools. There is a core ArchMapper component which implements the generic code generator and conformance checker as well as the user interface. It uses an internal representation of the architectural description, which is supplied by ADL-specific plug-ins. Currently, only the Acme plug-in is implemented, but UML support could be easily added as indicated

[6] Available for download at http://archmapper.sourceforge.net/

in the figure. Style-specific code generators and conformance checkers are also added via the Eclipse plug-in infrastructure, as it is currently done with the SpringStyleMapper plug-in.

3.3 ArchMapper Tool Features

In addition to the style-based architectural conformance check, the ArchMapper tool checks the following properties on the architectural level:

Average Component Dependency: The Average Component Dependency metric [7] is calculated and shown as a warning, independently from its value.

Dependency cycles: Components that are part of a dependency cycle are identified.

4 Evaluation

The ArchMapper tool has been evaluated using a relevant middleware-oriented architectural style of the Spring Model/View/Controller framework[7] and a real-world software system that has been in use at a postal service company.

4.1 Spring Web-MVC Architectural Style and Style-specific Mapping

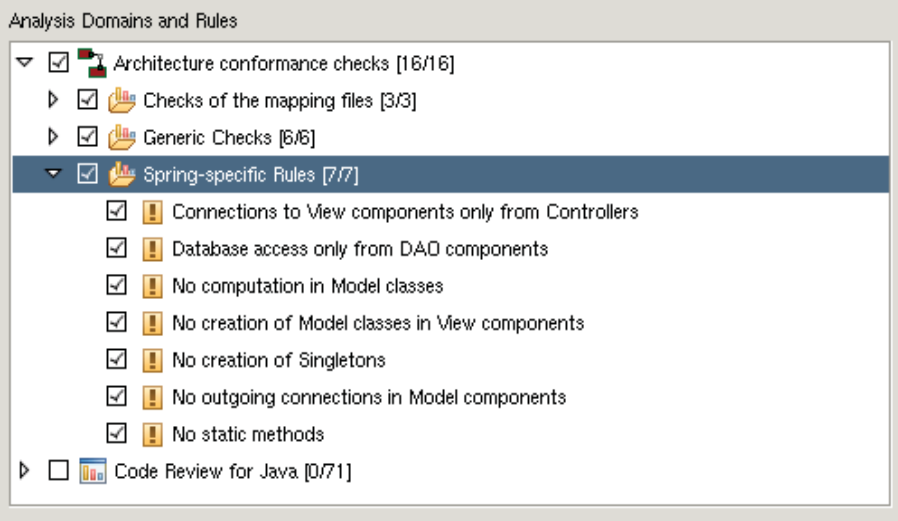

Fig. 3. Screenshot:Conformance check rules specific to the Spring Web-MVC style

[7] http://static.springsource.org/spring/docs/2.0.x/reference/mvc.html

Figure 3 shows the conformance check rules that have been defined specifically for the Spring MidArch Style, which is a specialisation of the general-purpose Model-View-Controller style.

One of the architectural rules is the "No computation in Model classes" rule. It is an architectural rule, since it governs the global interaction structure within the application. It may be stated on the architectural level, however it cannot be checked on the architectural level, but only using the implementation. In terms of the implementation elements, it means that classes within a Model component may only offer simple getter and setter methods.

The source code generation feature is used to generate the Spring XML configuration file from the architectural description.

4.2 Application to the Architecture of an Existing Software System

The ArchMapper tool identified several violations of architectural rules that have apparently not been noticed before. First, there are violations of the "No computation in Model classes" rule, which has been violated 40 times in the original implementation that was checked using ArchMapper. Additional violations of architectural rules that have been uncovered by ArchMapper include a large number of violations of the generic architectural rule of communicational integrity, for example direct accesses from classes of the controller component to database classes. In addition, violations of the rules "No outgoing connections in Model components" and "No static methods" have been detected. While the violations of the communicational integrity rule could have been detected with a generic architectural conformance checking tool, the other architectural rules could not have been stated easily without the unique style-based approach underlying ArchMapper, and hence the detection of their violation would not have been possible.

5 Related Work

5.1 Academic Approaches

There are several academic approaches to architectural conformance checking, but few that consider the relevance of style-specific checks. First, representative for those approaches that do not consider architectural styles, we discuss the ArchJava approach. Second, we discuss the approach supported by the tool ArchitectureChecker.

ArchJava Architectural information can be either provided as additional annotations that are part of the source code files, or it can be provided as a separate artefact. ArchJava [8] uses the former approach, while we chose the latter approach, which has several benefits:

– The software architecture has a different lifecycle than the code, since it is initially created before any code is written. Thus, it can be used for code generation. In addition, it is changed much less often than the code.

- A separate software architecture description can be easily used as architectural documentation. If the architectural information is distributed over a large number of source code files, relationships between different fragments are difficult to understand.
- Notations that are commonly used for architecture descriptions, e.g. the UML or specialised architecture description languages, often are graphical and are not designed to be intermixed with source code. A language for annotating source code would require an additional learning effort.
- The mapping between architecture and code is made explicit. It can be specified by rules and exceptions to these rules.

ArchitectureChecker In [9], another style-based approach to checking architectural conformance is presented. It is based on a declarative description of the architectural style. In difference to the approach presented here, it only uses the description of the style and a representation of the architecture derived from the source code, i.e. a module view. It does not use an explicit model of the expected architecture. It is not explicitly stated whether the architectural style refers to the component-and-connector view or to the module view. A tool implementing the approach also exists (ArchitectureChecker), but is not described in detail in the paper.

5.2 Industrial Tools

There exist some industrial tools that allow to perform architecture conformance checks. However, none of these tools supports the notion of an architectural style natively. Furthermore, they are not based on an independent architectural descriptions from the component-and-connector viewpoint, but rather operate on code-based architectural elements and their dependencies.

SonarJ[8] supports the definition of matrix-like structures of architectural elements and implicitly defines a kind of basic architectural style that limits legal dependencies between the architectural elements. It is limited to software systems implemented in Java.

Sotograph[9] is a tool with an academic background [10] which has been extended towards an industrial-strength tool. It is implicitly based on a sort of layered architectural style.

The CAST Application Intelligence Platform[10] also provides some capabilities for specifying and checking architectural rules.

6 Conclusions and Outlook

Compared to other approaches, ArchMapper has several unique properties:

[8] http://www.hello2morrow.com/products/sonarj
[9] http://www.hello2morrow.com/products/sotograph
[10] http://www.castsoftware.com/Product/Application-Intelligence-Platform.aspx

- ArchMapper is the only tool that is based on an architecture that can easily accommodate arbitrary architectural description notations, since we separate the architecture description language from the mapping approach.
- ArchMapper is one of only two existing approaches to exploit style information in the mapping.
  Our approach is more general, since the tool used by [11] is restricted to layered architectures.
- ArchMapper is the only approach that is designed to produce architectural rules and style-specific mappings across independently developed software systems, since the architectural styles and style-specifics mapping aspects are bound to the middleware product and platform that is used to build the software system.
- ArchMapper is integrated with a popular IDE (Eclipse) and provides information on architectural violations directly side-by-side with the source code.
- ArchMapper is the only approach that allows the addition of a style-specific conformance checker, and is furthermore integrated with a code generation feature, which has not been detailed in this paper.
- While the implementation is currently only available for Java as the implementation language, its architecture allows it to be easily be adapted to other implementation languages that are supported by the Eclipse Static Analysis Tools.

There are still some limitations to the approach, for which remedies could be provided by future work:

- When using Acme as an ADL, it is not possible to explicitly specify the direction of communication links. These are implicitly determined by the semantics of the defined ports. This could be relieved by using a different ADL for architecture specification, such as the UML-based modelling approach presented in [6].
- The mapping between the component-and-connector view and the module view does not yet accommodate for modules that are not associated with exactly one component, which is usually the case for any library module. One way to handle this is to define library pseudo-components for each library module. A component type for library components can be supplied by a specific architectural style. Style-specific rules are that library components may not access any non-library components. Among the library components, communication restrictions may also be explicitly defined. The mapping of library components to their source code may be reused for any software system using the libraries.
- While it is quite easy to adapt the tool to an implementation language other than Java, the application to multi-language software systems would require an additional effort to make the specification of the architecture-to-code mapping convenient.

References

1. Pahl, C., Giesecke, S., Hasselbring, W.: Ontology-based modelling of architectural styles. Information & Software Technology **51** (2009) 1739–1749
2. Kazman, R., Bass, L., Webb, M., Abowd, G.: SAAM: a method for analyzing the properties of software architectures. In: ICSE '94: Proceedings of the 16th international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1994) 81–90
3. ISO: Recommended Practice for Architectural Description of Software-Intensive Systems. (2006) IEEE Standard 1471-2000, ISO/IEC Standard 42010 (formerly ISO/IEC DIS 25961).
4. Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Pearson Education (2002)
5. Garlan, D., Shaw, M.: An introduction to software architecture. In Ambriola, V., Tortora, G., eds.: Advances in Software Engineering and Knowledge Engineering, Singapore, World Scientific Publishing Company (1993) 1–39
6. Giesecke, S., Marwede, F., Rohr, M., Hasselbring, W.: A Style-based Architecture Modelling Approach for UML 2 Component Diagrams. In: Proceedings of the 11th IASTED International Conference Software Engineering and Applications (SEA 2007), Cambridge, MA, USA, Anaheim, CA, USA, ACTA Press (2007) 530–538
7. Beck, C., Stuhr, O.: Stan – strukturanalyse für java. JavaSpektrum (2008) 44–49
8. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: Proceedings of the 24th international conference on Software engineering, ACM Press (2002) 187–197
9. Becker-Pechau, P.: Stilbasierte architekturprüfung. In Fischer, S., Mähle, E., Reischuk, R., eds.: Informatik 2009. Volume P-154 of Lecture Notes in Informatics., Bonn, Germany, Gesellschaft für Informatik e.V. (GI) (2009) 3264–3275
10. Bischofberger, W.R., Kühl, J., Löffler, S.: Sotograph - a pragmatic approach to source code architecture conformance checking. In Oquendo, F., Warboys, B., Morrison, R., eds.: Software Architecture, First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004, Proceedings. Volume 3047 of Lecture Notes in Computer Science., Springer (2004) 1–9
11. Becker-Pechau, P., Karstens, B., Lilienthal, C.: Automatisierte softwareüberprüfung auf der basis von architekturregeln. In Biel, B., Book, M., Gruhn, V., eds.: Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik, 28.-31.3.2006 in Leipzig. Volume 79 of LNI., GI (2006) 27–37