Science of Computer Programming 90 (2014) 135–160

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Semi-automated architectural abstraction specifications for supporting software evolution

Thomas Haitzer *, Uwe Zdun

Software Architecture Group, Faculty of Computer Science, University of Vienna, Vienna, Austria

HIGHLIGHTS

- We provide semi-automatic architecture abstractions based on UML-component models.
- Stable architecture abstractions with respect to software evolution.
- Built-in support for traceability and consistency checking.

ARTICLE INFO

Article history:
Received 15 October 2012
Received in revised form 11 October 2013
Accepted 12 October 2013
Available online 24 October 2013

Keywords:
Architectural abstraction
Architectural component and connector views
Software evolution
UML
Model transformation

ABSTRACT

In this paper we present an approach for supporting the semi-automated architectural abstraction of architectural models throughout the software life-cycle. It addresses the problem that the design and implementation of a software system often drift apart as software systems evolve, leading to architectural knowledge evaporation. Our approach provides concepts and tool support for the semi-automatic abstraction of architecture component and connector views from implemented systems and keeping the abstracted architecture models up-to-date during software evolution. In particular, we propose architecture abstraction concepts that are supported through a domain-specific language (DSL). Our main focus is on providing architectural abstraction specifications in the DSL that only need to be changed, if the architecture changes, but can tolerate non-architectural changes in the underlying source code. Once the software architect has defined an architectural abstraction in the DSL, we can automatically generate architectural component views from the source code using model-driven development (MDD) techniques and check whether architectural design constraints are fulfilled by these models. Our approach supports the automatic generation of traceability links between source code elements and architectural abstractions using MDD techniques to enable software architects to easily link between components and the source code elements that realize them. It enables software architects to compare different versions of the generated architectural component view with each other. We evaluate our research results by studying the evolution of architectural abstractions in different consecutive versions of five open source systems and by analyzing the performance of our approach in these cases.

1. Introduction

In many software projects the design and the implementation drift apart during development and system evolution [1]. In some small projects this problem can be avoided, as it might be possible to understand and maintain a well written

* Corresponding author. Tel.: +43 1 4277 78521; fax: +43 1 4277 8 78521.
E-mail addresses: thomas.haitzer@univie.ac.at (T. Haitzer), uwe.zdun@univie.ac.at (U. Zdun).
URLs: http://informatik.univie.ac.at/thomas.haitzer (T. Haitzer), http://informatik.univie.ac.at/uwe.zdun (U. Zdun).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.10.004

source code without additional architectural documentation. For many larger systems, this is not an option, and additional architectural documentation is required to aid the understanding of the system and especially to comprehend the "big picture" by providing architectural knowledge about a system's design [2]. One way to provide this information are automatically generated diagrams of the systems (e.g. in form of class diagrams) [3]. However these diagrams usually do not represent higher-level abstractions, and hence they hardly support the understanding of the big picture. First of all, the sheer size of the automatically generated diagrams is often a problem. In addition, creating an automatic layout or partitioning that is understandable is still an open research topic [4,5]. Clustering approaches from the reengineering research literature (e.g. [6–8]) can help to obtain an initial understanding and make sense of such diagrams. However the case study by Corazza et al. [9] shows that in five out of seven cases it is necessary to make manual corrections for about half of the entities of the analyzed source code.

As a consequence, today the documentation of the system's architecture is usually maintained manually. To model architectural knowledge, often models using box-and-line-diagrams [10], UML [11], architecture description languages (ADLs) [12], or similar modeling approaches are used. In many cases, such models are created before the actual implementation begins. Later, during implementation and system evolution, they loose touch with reality because changes to the software design are only made in the source code while the architectural models are not updated [13]. This problem is known as *architectural knowledge evaporation* [1].

Our approach focuses on architectural abstractions from the source code in a changing environment while still supporting traceability. It was initially introduced in a paper at the QoSA 2012 conference [14]. In this article we further extend our approach with respect to its traceability and consistency checking capabilities. We describe in detail how the approach provides these features to the software architect. Furthermore we provide extended case studies of our approach which also include additional information regarding traceability and consistency.

A considerable number of works exist that focus on abstractions from source code [15,16,8,17]. However, to the best of our knowledge, so far none of these approaches targets architectural abstractions at different levels of granularity, traceability between architectural models and the code, and the ability to cope with the constant evolution of software systems. Our approach introduces the semi-automatic abstraction of architectural component and connector view models from the source code based on an architectural abstraction specified in a domain specific language (DSL) [18,19]. In contrast to the related works, our approach specifically targets architectural abstractions and requires changes to the architectural abstraction specifications only in the rare case that the architecture of the system changes, but not for the vast majority of non-architectural changes we see during a software system's evolution (see Section 2). Please note that in the literature the term "component model" is often used to describe meta-models for component-based development [20]. In this paper, we (only) use the term *architectural* component and connector view (or component view for short) to describe a model that contains architectural components (as in [21]).

We chose a semi-automatic approach to enable the software architect to provide information which system details are relevant for getting the right level of abstraction – as software architecture is usually described in different views at different levels of abstraction. Our goal is to let the software architect specify this information with minimal effort in an easy-to-comprehend DSL that provides good tool support. Our approach allows architects to create different architectural abstraction specifications that represent different levels of abstraction and thus supports views ranging from high-level software architectural views to more low-level software design views. Once the software architect has defined an architectural abstraction in the DSL, we can automatically generate architectural component views from the source code using model-driven development (MDD) techniques and check whether architectural design constraints are fulfilled by these models.

As our approach focuses on defining stable abstractions in the architectural abstraction specification, it can cope with many changes to the underlying source code without changing the architecture description (i.e., an instance of the DSL). Only changes to the architecture itself, which usually require a substantial modification of the source code, require the architectural abstraction specification to be updated. By creating different versions of the architectural component view over time, we are able to use a delta comparison to check and reason about the changes of the architectural component view. The generated models can be compared to a design model, to check the consistency of an implementation and its design, and to analyze the differences. To support the iterative nature of our approach, it also supports automatically checking the consistency between the source code model and the architecture abstraction specification on the fly.

Once the architectural component views have been abstracted, another problem is to identify which parts of the source code contribute to a specific component, i.e., to support traceability between architectural models and code. Today, this usually requires substantial and non-trivial manual effort to identify which code elements are related to which model elements. In contrast, in our approach, traceability can be automatically ensured, as model-driven development (MDD) [22] is used to generate the required traceability links between the model elements and the source code directly from the architectural abstraction specification.

The remainder of the paper is organized as follows: Section 2 explains the research problem addressed by this paper in more detail, as well as the research method that was applied to design and evaluate the DSL. Section 3 gives an overview of our approach. Section 4 provides details about our architectural abstraction DSL and its implementation. In Section 5 we present the evaluation of our approach based on five cases and a performance evaluation. In Section 6 we discuss open issues and lessons learned. Section 7 compares our approach to the related work, and we conclude in Section 8.