Information and Software Technology xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

The WGB method to recover implemented architectural rules

Vanius Zapalowski[*,a], Ingrid Nunes[a,b], Daltro José Nunes[a]

[a] Instituto de Informática, UFRGS, Porto Alegre, Rio Grande do Sul, Brazil
[b] Department of Computer Science, Technische Universität Dortmund, Dortmund, Germany

ARTICLE INFO

Keywords:
Software architecture
Architectural rule
Source code dependency
Architecture recovery

ABSTRACT

Context: The identification of architectural rules, which specify allowed dependencies among architectural modules, is a key challenge in software architecture recovery. Existing approaches either retrieve a large set of rules, compromising their practical use, or are limited to supporting the understanding of such rules, which are manually recovered.

Objective: To propose and evaluate a method to recover architectural rules, focusing on those implemented in the source code, which may differ from planned or conceptual rules.

Method: We propose the WGB method, which analyzes dependencies among architectural modules as a graph, adding weights that correspond to the proposed *module dependency strength* (MDS) metric and identifies the set of implemented architectural rules by solving a mathematical optimization problem. We evaluated our method with a case study and an empirical study that compared rules extracted by the method with the conceptual architecture and source code dependencies of six systems. These comparisons considered efficiency and effectiveness of our method.

Results: Regarding efficiency, our method took 45.55 s to analyze the largest system evaluated. Considering effectiveness, our method captured package dependencies as extracted rules with a reduction of 87.6%, on average, to represent this information. Using allowed architectural dependencies as a reference point (but not a gold standard), provided rules achieved 37.1% of precision and 37.8% of recall.

Conclusion: Our empirical evaluation shows that the implemented architectural rules recovered by our method consist of abstract representations of (a large number of) module dependencies, providing a concise view of dependencies that can be inspected by developers to identify occurrences of architectural violations and undocumented rules.

1. Introduction

Software architecture recovery has been largely investigated to support the development of software systems, which often have missing or outdated architectural documentation [1]. This recovered information helps understand the software structure as well as rules that govern the interaction among its modules. The lack of (an updated) documented knowledge regarding the architecture causes a disorganized software evolution, which leads to major maintenance problems [2,3]. For example, the introduction of architectural violations leads to the known problems of *architecture drift* and *architecture erosion* [4]. Moreover, the inspection of implemented architectural rules may reveal unplanned rules introduced by developers, which typically remain undocumented [5]. Consequently, retrieving this kind of architectural information mitigates the *knowledge vaporization* [6] problem.

The support that architecture recovery approaches provide varies in nature. They can, for example, identify architectural modules [7–10] by clustering software elements, e.g. classes, when the implemented software structure is inconsistent with the code. With respect to architectural rules, visualizations [11,12] have been developed to help developers understand rules that are implemented in the code. Moreover, other solutions aim to automatically recover implemented rules, but they are restricted to limited scenarios. For example, rules mined by the PR-Miner [13] tool are limited to coding rules among procedures and functions implemented with the procedural paradigm, and the approach proposed by Hora et al. [14,15] focuses only on patterns related to external APIs.

In this paper, we focus on recovering rules that specify allowed dependencies between architectural modules. Such recovered rules are those implemented, i.e. those reflecting what is actually present in the code, which may be inconsistent with rules that are in developers' mindset or documentation. Our proposal consists of a three-step method, named the *weighted-graph-based* (WGB) method, which identifies implemented rules by means of the construction of a weighted

* Corresponding author.
E-mail addresses: vzapalowski@inf.ufrgs.br (V. Zapalowski), ingrid.nunes@inf.ufrgs.br (I. Nunes), daltro@inf.ufrgs.br (D.J. Nunes).

https://doi.org/10.1016/j.infsof.2018.06.012
Received 10 August 2017; Received in revised form 8 May 2018; Accepted 22 June 2018

graph using as input source code dependencies and their posterior analysis. In short, for every module pair, we calculate the *module dependency strength* (MDS) metric, which represents how much a module depends on another, considering its sub-modules and surrounding modules. Therefore, our proposed metric is not limited to counting method calls, but takes into account many factors, such as the number of sibling modules and usage ratios. This metric is used as weights of a graph in which nodes represent modules and arrows dependencies. This graph has some of its arrows removed based on a pairwise analysis of sets of modules, and then an optimization problem is solved to give the recovered implemented architectural rules.

2. Definitions and problem

There are many ways of representing a software architecture and specifying architectural rules. In our work, we assume that the architecture is decomposed into modules, which may be further refined into sub-modules, leading to a module hierarchy in the form of a tree. This is often the way that modules are implemented or expressed in a documented software architecture. Each (sub-)module may contain software elements, e.g. classes. Architectural rules explicitly specify dependencies between pairs of modules, each indicating *allowed* dependencies between elements of the modules, e.g. a method call. The representation of a software architecture is thus composed of a module hierarchy (structured as a tree) and rules connecting (sub-)modules, forming a graph.

Rules and dependencies are illustrated in Fig. 1, which shows an architecture with three main layered modules, each with sub-modules. Thick arrows represent architectural rules, indicating for example that the *Presentation* module depends on the *Business* module, meaning that any element of the former can depend on any element of the latter. We represent such a rule, or module dependency, as Presentation → Business.

Architectural rules have additional implications. First, rules consider not only the elements that are within the module referred in the rule, but also any element in the module hierarchy. This means that, for example, any element of the *Presentation's* sub-modules can depend on any element of the *Business's* sub-modules. Second, dependencies between non-represented sub-modules of a module are allowed. Thus, dependencies between omitted sub-modules of the *Business Objects* module are permitted. Finally, everything else is *forbidden* and, if occurs, is an *architectural violation*, e.g. the dependency between the *Feature Z* and *Schema Z DAO* modules shown in Fig. 1. We are aware that there are many sophisticated languages to express more refined constraints over architectural modules [16–18]. However, documented architectural models are typically as simple as detailed in our description [19].

When a software system is implemented, it may have different, possibly diverging, architecture representations. The first is the representation that is *expected* to be in the code, referred to as *conceptual* architecture. It may be (1) an architecture model that is documented as a planned architecture, (2) only available in the developers' mindset, or (3) both—and these may be inconsistent if, for example, the documentation is outdated. The second representation is the one that is consistent with the source code, referred to as *implemented* architecture. A possible model of the implemented architecture can consist of all the dependencies that occur in the code among modules. However, even in small-scale software projects, this would result in an illegible model, due to low granularity level. Divergences between the conceptual and implemented architectures can be either architectural violations or undocumented rules.

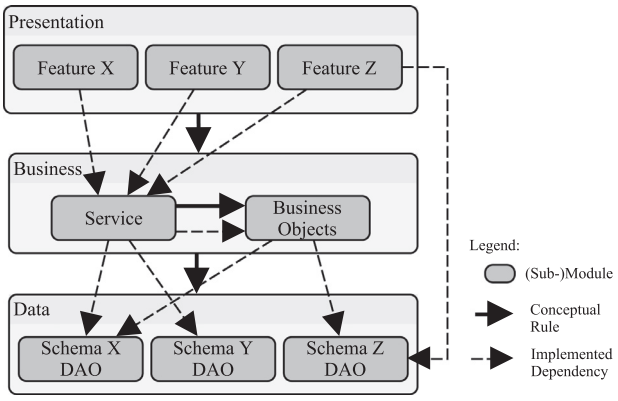

Fig. 1. Conceptual architectural rules *vs.* implemented dependencies.

Our goal in this work is to *recover implemented architectural rules* using a given module hierarchy with elements assigned to modules as well as dependencies between elements present in the source code. This module hierarchy can be manually specified, derived from the use of clustering techniques or an implemented software organization. Modules to be documented in the software architecture match a certain level of this organization, hierarchically structured as a tree, e.g. in terms of packages. However, this organization may include super-modules due to code conventions, like in Java, or sub-modules that are too fine-grained to be represented in the architecture. In the remainder of this paper, we adopt the implemented module hierarchy to exemplify and evaluate our method because it does not require an input other than the source code.

Given this module hierarchy, we must identify rules that reflect dependencies in the code at the right level of granularity. This means that *rules must be expressed at the highest granularity level as possible*. For example, if there are many dependencies between the *Business's* and *Data's* sub-modules, they must be abstracted to a rule Business → Data. This abstraction has two implications, detailed as follows.

1. *Implemented rules may capture hidden information, not expressed in conceptual rules.* Dependencies from the *Presentation* to the *Business* module are, in fact, to the *Service* sub-module. Diverging from the conceptual rule, the most appropriate implemented rule should be Presentation → Service.
2. *Rules associated with sparse dependencies must be fine-grained.* Sparse or infrequent dependencies are those that occur in the code but are not frequent enough to be generalized to a coarse-grained rule, i.e. in terms of super-modules. Such dependencies may be architectural violations, e.g. dependencies from the *Presentation's* to a *Data's* sub-module. Therefore, in this case, it is desired that these dependencies are not abstracted to a general rule (e.g. Presentation → Data), but remain fine-grained (e.g. Feature X→ Schema Z DAO), so that it can be identified as a violation, when recovered implemented architectural rules are inspected by developers.

Considering these introduced definitions and specified problem, we proceed to the presentation of our method proposed to recover implemented architectural rules.

3. WGB method

Our WGB method chooses *a set of architectural rules to represent an implemented software architecture taking as input a given software module organization (e.g., package structure) and dependencies among module elements* (e.g., classes). These recovered rules are a coarse-grained representation of the implemented architecture that are an architectural view of the system. Our method is composed of three sequential steps: (i) calculation of a metric that captures the dependency strength between every two modules, considering dependencies between elements (Section 3.1); (ii) pairwise clusterization of dependencies based on this metric, considering neighbor module levels (Section 3.2); and (iii) selection of the set of rules that maximizes the dependency strength without redundancy, which we assume as the correct rule granularity to