The Journal of Systems and Software 125 (2017) 15–34

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Exploiting traceability uncertainty between software architectural models and extra-functional results

Catia Trubiani[a,*], Achraf Ghabi[b], Alexander Egyed[c]

[a] Gran Sasso Science Institute, L'Aquila, Italy
[b] celum GmbH, Linz, Austria
[c] Johannes Kepler University, Linz, Austria

ARTICLE INFO

Article history:
Received 15 February 2016
Revised 29 September 2016
Accepted 23 November 2016
Available online 23 November 2016

Keywords:
Traceability
Uncertainty
Software modeling
Extra-functional results

ABSTRACT

Deriving extra-functional properties (e.g., performance, security, reliability) from software architectural models is the cornerstone of software development as it supports the designers with quantitative predictions of system qualities. However, the problem of interpreting results from quantitative analysis of extra-functional properties is still challenging because it is hard to understand how the analysis results (e.g., response time, data confidentiality, mean time to failure) trace back to the architectural model elements (i.e., software components, interactions among components, deployment nodes).

The goal of this paper is to automate the traceability between software architectural models and extra-functional results, such as performance and security, by investigating the uncertainty while bridging these two domains. Our approach makes use of extra-functional patterns and antipatterns, such as performance antipatterns and security patterns, to deduce the logical consequences between the architectural elements and analysis results and automatically build a graph of traces, thus to identify the most critical causes of extra-functional flaws. We developed a tool that jointly considers SOftware and Extra-Functional concepts (SoEfTraceAnalyzer), and it automatically builds model-to-results traceability links. This paper demonstrates the effectiveness of our automated and tool supported approach on three case studies, i.e., two academic research projects and one industrial system.

1. Introduction

In the software development domain there is a very high interest in the early validation of extra-functional requirements because this ability avoids late and expensive repairs to consolidated software artifacts (Chung et al., 2012). One of the proper ways to manage software quality is to systematically predict the extra-functional properties of the software system throughout the development process. It is thus possible to make informed choices among architectural and design alternatives; and knowing in advance if the software will meet its extra-functional objectives (Grunske et al., 2007).

Advanced Model-Driven Engineering (MDE) techniques have successfully been used in the last few years to introduce automation in software quality modeling and analysis (Ameller et al., 2010). Nevertheless, the problem of interpreting extra-functional results is still quite challenging. A large gap exists between the representation of extra-functional analysis results and the software architectural model provided by the engineers. In fact, the former usually contains numbers (e.g., throughput variance, vulnerability level, mean time to failure, etc.), whereas the latter embeds architectural choices (e.g., software components, interaction among components, deployment nodes). Today, the interpretation of extra-functional results is mostly based on the analysts' experience and therefore its effectiveness often suffers from lack of automation (Woodside et al., 2007).

In Ghabi and Egyed (2015) we proposed a language capable of capturing model-to-code traceability while considering typical uncertainties in its domain. For example, the engineer knows that some given piece of code may implement an architectural element; however, not whether this piece of code also implements other architectural elements; or whether this architectural element is also implemented elsewhere (other code). This paper adapts this language to provide model-to-results traceability links while considering typical uncertainties from the extra-functional analysis domain. We presume that engineers know when a given extra-functional result is affected by an architectural element. However, they may not know whether this extra-functional result is also affected by

* Corresponding author.
E-mail addresses: catia.trubiani@gssi.infn.it (C. Trubiani), a@ghabi.net (A. Ghabi), alexander.egyed@jku.at (A. Egyed).

http://dx.doi.org/10.1016/j.jss.2016.11.032
0164-1212/

other architectural elements or whether other extra-functional results are also affected by this architectural element.

Further knowledge can be considered to better understand the relationship between architectural elements and extra-functional results, in particular extra-functional patterns and antipatterns (Vlissides et al., 1995; Brown et al., 1998) represent best and bad practices in architectural models affecting extra-functional properties. A pattern specification (Vlissides et al., 1995) includes solutions to commonly occurring problems, e.g., security patterns (Fernandez-Buglioni, 2013) encapsulate knowledge and expertise to improve security properties such as confidentiality, integrity, etc. An antipattern definition (Brown et al., 1998) includes the description of bad practices occurring in the architectural model along with the solution that can be applied to avoid negative consequences, e.g., performance antipatterns (Smith and Williams, 2012) collect domain-expert knowledge to react against performance flaws such as low response time, high network utilization, etc.

This paper is an extension of Trubiani et al. (2015) where we focused on performance analysis results and we used software performance antipatterns to reduce model-to-results traceability uncertainties. The contribution of this paper is to provide support in the process of identifying the architectural model elements that most likely contribute to the violation of multiple extra-functional requirements by jointly considering knowledge from engineers and extra-functional patterns and antipatterns. To this end, we developed a tool, namely SoEfTraceAnalyzer Trubiani et al. (2016), that jointly considers Software and Extra-Functional concepts: it takes as input a set of statements specifying the relationships between software elements and extra-functional properties, and provides as output model-to-results traceability links. The language defined in Ghabi and Egyed (2015) is extended by adding a weighting methodology that quantifies the extra-functional requirements' violation, thus to highlight the criticality of model elements despite extra-functional properties. The key feature of our tool is that the knowledge of extra-functional patterns and antipatterns can be embedded in the specification of uncertainties to deduce the logical consequences between architectural elements and analysis results, thus to disambiguate the limited knowledge of engineers.

Our approach is not limited, in principle, to specific extra-functional properties. However, to investigate the effectiveness of traceability links, in this paper we decided to focus on performance and security, and we make use of security patterns (Fernandez-Buglioni, 2013) and performance antipatterns (Smith and Williams, 2012) to reduce traceability uncertainty. This choice is driven by the fact that security has a "direct" overhead on performance, whereas other extra-functional properties may not. For example, it is well known that introducing security mechanisms, such as encryption of data, inevitably consume system resources influencing the system performance, even affecting its full operability. On the contrary, increasing system reliability may mean to create copies of software components off-line without any impact on the system performance.

The paper is organized as follows: Section 2 describes an illustrative example; Section 3 discusses the relationships between software development artifacts and extra-functional properties; Section 4 describes our approach; Section 5 illustrates the validation of the approach on three case studies (i.e., two academic research projects and one industrial system); Section 6 discusses the threats to validity of the approach; Section 7 presents related work; Section 8 concludes the paper.

2. Illustrative example

In this section we illustrate the *DesignSpace*, an academic research project aimed at building an engineering infrastructure to integrate diverse development artifacts and their relations (Demuth et al., 2015). It is an engineering platform for the exchange, linking, and validation of the knowledge across different artifacts. It supports distributed collaboration, a wide range of tools and development, maintenance, and evolution of services including incremental consistency checking and transformation. There are three main challenges: (i) how the knowledge created and manipulated by engineers in their respective single-user tools is being made available to other engineers; (ii) how this knowledge is interconnected to express cross-tool dependencies; (iii) how engineers benefit from analysis and transformation techniques.

Fig. 1 provides a high-level overview of the DesignSpace system (Demuth et al., 2015) to illustrate some examples of the involved artifacts and their traceability, thus to demonstrate the need of our approach. Fig. 1a provides a high-levelcrs and artifacts involved in a system. There are five engineers: Alice is an architect and collaborates with Bob on the modeling of a subpart of the system reported in the software architectural model m_1. Bob is also an analyst and checks if the extra-functional results r_2 of this subsystem are fulfilling the stated requirements. Similarly, Carol is an architect and collaborates with David on the modeling of an other subpart of the system reported in the model m_3. David is also an analyst and checks the results r_4 of this last subsystem. Finally, Paul is a project manager and manages the software architectural model m_5 modeling the whole system, and the global extra-functional results r_6. Examples of both functional and extra-functional requirements are reported in Fig. 1b: R_1 is a functional requirement through which the architectural models are updated, e.g., changes to model m_1 require modifications to m_5 but not to m_3; R_2 is a performance requirement requesting a maximum delay of 3 s when loading the models; R_3 is a security requirement regulating the users access to models under authorization, e.g., Alice can access to model m_1 only, David can access to model m_3 and results r_4, whereas Paul can access to all artifacts, i.e., all models and extra-functional results.

As stated in Egyed and Grünbacher (2004), when analyzing trade-offs among these requirements, the developer must understand how the requirements affect each other. The goal of our approach is to trace extra-functional results to architectural model artifacts thus to support software designers in the task of identifying the most suitable model elements responsible for bad properties, if any.

3. Software architectural models and extra-functional properties

Common practice for software engineers is to document architectural descriptions, however it is less common to document how such architectural elements (e.g., software components, dynamic scenarios, deployment nodes, etc.) are related to extra-functional properties (e.g., performance, security). Knowing about traceability is important to understand what are the architectural elements contributing to extra-functional properties and deriving the most suitable refactoring actions to improve such properties. The goal of this work is to support software designers in the task of identifying the relationships between software architectural models and extra-functional results.

We refer to specific software architectural elements where the granularity of an architectural element is entirely user-definable. An architectural element could be a software component, a service built on top of several components, or any other logical grouping (e.g., a hardware device hosting several software components). We will discuss the implications of different granularity choices later.

We refer to specific extra-functional properties, such as performance (e.g., response time, throughput, utilization) and security (e.g., security level/risk). Here also the granularity is arbi-