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

Catia Trubiani\textsuperscript{a,}\textsuperscript{,} Achraf Ghabi\textsuperscript{b}, Alexander Egyed\textsuperscript{c}

\textsuperscript{a}Gran Sasso Science Institute, L’Aquila, Italy
\textsuperscript{b}celum GmbH, Linz, Austria
\textsuperscript{c}Johannes Kepler University, Linz, Austria

\textbf{A R T I C L E   I N F O}

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

\textbf{A B S T R A C T}

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.

© 2016 Elsevier Inc. All rights reserved.

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 (Grunsk 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 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
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 SoEffTraceAnalyzer (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-level view of artifacts involved in a system. There are five engineers: Alice is an architect and collaborates with Bob on the modeling of a subsystem of the system reported in the software architectural model m1. Bob is also an analyst and checks if the extra-functional results r2 of this subsystem are fulfilling the stated requirements. Similarly, Carol is an architect and collaborates with David on the modeling of another subsystem of the system reported in the model m3. David is also an analyst and checks the results r4 of this last subsystem. Finally, Paul is a project manager and manages the software architectural model m2 modeling the whole system, and the global extra-functional results r5. Examples of both functional and extra-functional requirements are reported in Fig. 1b: R1 is a functional requirement through which the architectural models are updated, e.g., changes to model m1 require modifications to m1 but not to m3; R2 is a performance requirement requesting a maximum delay of 3 s when loading the models; R3 is a security requirement regulating the users access to models under authorization, e.g., Alice can access to model m1 only. David can access to model m3 and results r4, 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-
The relationship between architectural elements and extra-functional properties is bidirectional. We expect that a single model element is contributing in multiple extra-functional properties (one-to-many mapping), for example a software component may affect the response time of a service and the utilization of its hosting hardware device. Similarly, an extra-functional property is affected by multiple model elements, but it may happen that the architectural specification is incomplete either by choice or by omission. For example, system response time and/or security level are two properties that may span the entire system, and they can be affected by parts of the system architecture that are not included as modeling artifacts. Consequently, it becomes very difficult to select specific model elements, besides others, that are not contributing to system properties.

4. Our approach

Fig. 2 illustrates the process we envisage to automate the traceability between architectural model elements and extra-functional results. Ovals in the figure represent operational steps whereas square boxes represent input/output data. Four main phases are identified in the process.

Phase ①: Requirements. We assume that a set of extra-functional requirements is defined. Some examples of performance requirements are as follows: the response time of a service has to be less than 3 s, the throughput of a service has to be greater than 10 requests/second, the utilization of a hardware device shall not be higher than 80%, etc. Some examples of security requirements are: a message exchanged between two components might require encryption depending on whether the communication channel between the components is a wireless one or not, the completion of a certain software operation needs to be verified by a certification authority, etc. All extra-functional (e.g., performance and security) requirements will be used to build the software architectural model and interpret the results obtained from the model-based analysis (Dobrica and Niemelä, 2002).

Phase ②: Modeling. In the modeling phase, an annotated software architectural model is built. Such model embeds the stated extra-functional requirements, e.g., to enable reliability properties some software components need to be duplicated, or to achieve a certain security level some software services have to be protected (e.g., a software component requiring a certain service must be authenticated before allowing its usage).

Phase ③: Analysis. In the analysis phase, extra-functional model(s) are obtained through model transformation, and such models (e.g., queuing networks for performance (Lazowska et al., 1984), and fault-trees for reliability (Lyu et al., 1996)) are solved to obtain the results of interest. Some extra-functional requirements (e.g., security) consist in the addition of appropriate mechanisms implementing the corresponding requirement in the software architectural model. As an example, if a security requirement specifies that data integrity must be guaranteed for a certain service, an additional pattern with the steps needed for the data integrity mechanism must be introduced in the architectural model whenever the service is invoked. All these additions are included into extra-functional model(s), thus to get results reflecting them.

Phase ④: Feedback. The focus of this paper is on the feedback phase where the extra-functional requirements must be in-

1 Annotations are aimed at specifying information to execute performance analysis such as the incoming workload, service demands, hardware characteristics, etc.
terpreted in order to detect, if any, extra-functional flaws.\textsuperscript{2} Such flaws may be of different nature and several refactoring actions can be introduced to improve the corresponding violated property. For example, to increase the reliability of a software component the number of its replications can vary, or to increase the security of a software service the key size of the encryption algorithm can be modified. This leads to a trade-off analysis that is conducted by implementing the available refactorings as new software architectural models that undergo the same process shown in Fig. 2.

The goal of our approach is to trace architectural model elements vs extra-functional results (see shaded boxes of Fig. 2), thus to highlight the architectural model elements contributing to extra-functional flaws. It starts with an automatic trace generation operational step that provides as output a weighted footprint graph (from input), i.e., a graph containing a node for every result element (called RE nodes) and a node for each model element (called ME nodes). The connections between these nodes describe the certainties of the input (trace or no-trace), and are refined with an automatic uncertainty reduction operational step aimed at generating a weighted footprint graph, i.e., after (anti)pattern-based rules. This latter step is supported by extra-functional patterns/antipatterns, because they suitably fit with our goal. In Trubiani et al. (2015) we used performance antipatterns (Smith and Williams, 2012) to deduce the logical consequences of the uncertainties, and we found that they nicely contribute to automatically generate traces joining architectural elements and performance results. In this paper we further consider security patterns (Voder and Barcalow, 1998) to enable a trade-off analysis among these two extra-functional properties.

4.1. Automatic trace generation

The automatic trace generation operational step (see Fig. 2) takes as input: (i) extra-functional requirements, (ii) annotated software architectural model, and (iii) extra-functional results. It provides as output a weighted footprint graph.

Extra-functional requirements are classified on the basis of the property (e.g., performance, security) they address and the level of abstraction they apply. In general, it is very challenging to quantify security-related properties, hence we leave to engineers the decision of associating pre-defined security levels (SEC), i.e., low, medium, and high. For performance we consider the requirements that refer to the following performance indices (Jain, 1991): Response time (RT) is defined as the time interval between a user request of a service and the response of the system; Throughput (TH) is defined as the rate at which requests can be handled by a system, and is measured in requests per unit of time; Utilization (U) is defined as the ratio of busy time of a resource and the total elapsed time of the measurement period; Queue length (QL) is defined as the number of users waiting for a resource; Waiting time (WT) is defined as the time interval required to access to a resource starting from when the resource is required up to when it is accessed.

Usually, RT requirements are upper bounds defined in “business” requirements by the end users of the system. TH requirements can be both “business” and “system” requirements, they can represent either an upper or a lower bound. U, QL and WT requirements are upper bounds defined in “system” requirements by system engineers on the basis of their experience, scalability issues, or constraints from other concurrent software systems.

Various levels of abstraction can be defined for a requirement: system, processor, etc. However, we do not consider all possible combinations of indices and levels of abstraction, we focus on the most common ones that are: SEC, RT, and TH of services, U, QL, and WT of hardware devices.

\textsuperscript{2} A flaw originates from a set of unfulfilled requirement(s), such as “the estimated average response time of a service is higher than the required one”, or “the achieved security level of a service is lower than the required one”.

Fig. 2. Deriving automatically model-to-results traceability links by means of extra-functional patterns/antipatterns.
Extra-functional results represent the analysis values of the indices we consider for traceability. Note that such values are affected by a set of features such as system workload and operation profile that represent how the software system is used (Jain, 1991).

Annotated software architectural models may be constituted by elements belonging to different views (Clements et al., 2003): Static/Software View (SW) includes the software elements, e.g., operations (SWop), components (SWcomp), services, and the static relationships among them; Dynamic/Interaction View (DY) specifies the interaction, e.g., messages (DYmsg), that occurs between the software components to provide services; Deployment/Hardware View (HW) includes the hardware devices, e.g., processing nodes (HWnode), and communication networks (HWnet), and the mapping of software components and interactions onto hardware devices. Summarizing, SWop, SWcomp, DYmsg, HWnode, and HWnet represent the architectural elements we consider for traceability.

4.1.1. Language for expressing traceability

This paper adopts the language for model-to-code traceability introduced in Ghabi and Egyed (2015) and extends it to express model-to-results traceability considering some of the unique aspects of this domain. The main benefit of our approach is that our language allows the engineer to express uncertainty constructs to the level of detail she or he is comfortable with. As drawback, neither correctness nor completeness are guaranteed.

Each construct is defined as \( m^r \) relationship \( r^r \) where \( m^r \) is the set of model elements and \( r^r \) is the set of results elements. The star symbol \( * \) expresses multiplicity in that \( m^r \) stands for multiple model elements and \( r^r \) for multiple results elements. The relationship term declares how the first set is related to the second one.

We distinguish between three major relationships: affectAtLeast, affectAtMost, affectExactly.

1) **AffectAtLeast Construct**: the input \( m^r \) affectAtLeast \( r^r \) defines that the model elements in \( m^r \) affect all of the result elements in \( r^r \) and possibly more. This input has a correctness constraint ensuring that every model element in \( m^r \) individually must be affecting a subset of \( r^r \). One example of this relationship is provided by the software components SWcomp and the subset of operations SWop involved in a service \( S \) that affect at least the security level (SEC), the response time (RT) and the throughput (TH) of the service \( S \).

Input: \{SWop, SWcomp\} affectAtLeast \{SEC, RT, TH\}

2) **AffectAtMost Construct**: the input \( m^r \) affectAtMost \( r^r \) defines that the model elements in \( m^r \) affect some of the result elements in \( r^r \) but certainly not more. This input expresses the certainty that every other model element not in \( m^r \) must not affect any result element in \( r^r \). One example of this relationship is provided by the software components SWcomp and the subset of operations SWop involved in a service \( S \) as well as the deployment nodes HWnode where the SWcomp components are deployed that affect at most the security level (SEC), the response time (RT) and the throughput (TH) of the service \( S \).

Input: \{SWop*, SWcomp*, HWnode\} affectAtMost \{SEC, RT, TH\}

3) **AffectExactly Construct**: the input \( m^r \) affectExactly \( r^r \) defines that every model element in \( m^r \) affects one or more result elements in \( r^r \) and that the results elements in \( r^r \) are not affected in any other model element not in \( m^r \). This input defines no-trace between each result element in \( r^r \) and each model element in the remaining \( M - m^r \) (where \( M \) is the set of all input model elements), since each model element in \( m^r \) affects only a subset of \( r^r \). However, this does not mean that these result elements could not be affected by other model elements in \( M - m^r \). One example of this relationship is provided by an hardware device HWnode and the performed operations SWop that affect exactly its utilization (U).

Input: \{SWop\}, HWnode\} affectExactly \{U\}

4.1.2. Weighted footprint graph

The language we provided to express the uncertainty constructs between a set of architectural model elements and a set of analysis results elements is very flexible. Listing 1 reports one abstract example for the specification of the input. For example, the hardware devices HWnode and HWnet affect exactly the performance indices related to them, i.e., utilization (U), queue length (QL), and waiting time (WT). As another example, the software components SWcomp and the subset of operations SWop involved in a service \( S \) affect at least the security level (SEC), the response time (RT) and the throughput (TH) of the service \( S \).

The goal of our SoFTraceAnalyzer tool (Trubiani et al., 2016) is to interpret these traceability expressions and automatically build (certainties and uncertainties) in a graph structure, which we call the weighted footprint graph (from input).

Fig. 3 reports one abstract example of this graph and it refers to the input specified in Listing 1. The graph contains a node for every result element (called RE nodes) and a node for each model element (called ME nodes). RE nodes are: response time (RT), throughput (TH), utilization (U), queue length (QL), waiting time (WT), and security level (SEC). ME nodes are: software operations (SWop), software components (SWcomp), dynamic interactions (DYmsg), hardware nodes (HWnode), and communication networks (HWnet).

The connections between RE nodes and ME nodes describe the certainties of the input (trace or no-trace) which are generated out of the logical consequences of the uncertainties. A trace (m, r) is depicted by a bold line between the ME node of m and the RE node of r. In Fig. 3 no such lines are depicted because the logical interpretation of the input did not yield any traces. On the contrary, no-traces are depicted by dashed lines. Furthermore, the graph contains nodes to capture model element groups (MEG nodes) and results element groups (REG nodes). These two kinds of nodes describe the uncertainties of the input.

Note that each result element node RE has a weight (\( \omega \)) that represents a value indicating how much the requirement is far from the analyzed index, whereas each model element node ME has a weight that is a function (\( \Sigma F(\omega) \)) indicating how much the architectural element is critical for the violated requirements. Our language allows to set the weights for RE nodes and their guilt estimates vs ME nodes, thus to quantify how much extra-functional results are affected by architectural elements. Listing 2 reports a simple example for the specification of weight and guilt estimates: the response time (RT) and security (SEC) results are violated by 0.5 and 0.33, respectively; the software component (SWcomp) participates to both results with different guilt estimates, i.e., 0.42 and 0.15 for RT and SEC, respectively. The numerical values provided as weight estimates come from the evaluation of stated extra-functional requirements, e.g., if the response time of a service is required to be not larger than 8 s, but the performance analysis reveals that the actual value is 12 s, then 0.5 represents the percentage of requirement violation. For security results it is indeed challenging to provide weights expressing requirement violation, hence we assume that security concerns can be discretized in a set of samples on the basis of their severity and numerical values.
are associated to these samples (e.g., security levels may be: high → 0.75, medium → 0.5, and low → 0.25). In this way, similarly to performance, we can report the estimation of requirement’ violation expressed as percentage among the specified values.

Guilt estimates for performance results are assigned by using the guilt-based approach we validated in our previous work (Trubiani et al., 2014a). In particular, each model element is ranked on the basis of how much it contributes to the performance index under analysis: we calculate the index of the corresponding model element and we estimate how much it is contributing. For security results it is indeed challenging to provide guilt estimates, hence we assume that security concerns can be associated to a set of model elements and numerical values are associated by ranking them, similarly to performance.

Fig. 4 shows the weighted footprint graph that is generated from input (see Listing 1) augmented with weight and guilt estimates specification reported in Listing 2. Note that the weights for RE nodes are directly derived from their specification, whereas weights for ME nodes are calculated while taking into account traceability information. If there exists a trace (m, r) then we assume that m fully participates to the weight of r, whereas if there is an uncertain trace then guilt estimates are used to scale the weight of r. Weights of model elements are obtained by summing up all traces and uncertain ones. In our example the SWcomp node is weighted by the following calculation: 0.5 × 0.42 + 0.33 × 0.15 = 0.26, in fact that node is involved with uncertain traces in both SEC and RT results.

Different heuristics (ω) and functions (Σ F(ω)) can be used to weight RE and ME nodes in footprint graphs. Furthermore, the human intervention of engineers may help to add priorities to extra-functional properties and to specify legacy constraints for architectural elements.

### 4.2. Automatic uncertainty reduction

The weighted footprint graph is the foundation for automatic trace generation, and several propagation rules can be introduced to reduce the initial uncertainty. Our approach makes use of performance antipatterns (Smith and Williams, 2003) and security patterns (Konrad et al., 2003) to deduce the logical consequences between architectural elements and analysis results.

(Anti)patterns represent a well-known technique to express domain-independent knowledge and expertise in a reusable way. Architectural and design patterns constitute solid solutions that can be employed in order to solve known and recurrent problems. (Anti)patterns provide the main advantage of including benefits and drawbacks in their specification, and they can be taken into account while sketching their solution.

#### 4.2.1. Performance antipatterns

Performance antipatterns deal with the performance issues of software systems. In literature, the main source of performance antipatterns is the work done across years by Smith and Williams (2012) that have ultimately defined a number of 14 notation- and technology- independent antipatterns. The benefit of using these antipatterns is that they offer solutions in forms of alternative architectures to performance flaws, and several works recently have demonstrated their usefulness (Chen et al., 2015; Trubiani et al., 2014b; Chen et al., 2014; Sharma and Anwer, 2013).

In our previous work (Cortellessa et al., 2014) we provided a logic-based representation of performance antipatterns that supports the specification of further input to trace generation. Listing 3 reports the traceability rules while considering the specification of some performance antipatterns, i.e., Concurrent Processing Systems (CPS), Pipe & Filter (P&F), Cod Class/Component (BLOB), Extensive Processing (EP), Empty Semi Trucks (EST), One-Lane Bridge (OLB), and The Ramp (TR), respectively.

For example, detecting a CPS antipattern indicates that HWndoe affects exactly QL and U. This rule comes from the logic-based formula of the CPS antipattern that has been defined in Cortellessa et al. (2014), and an excerpt is reported in Eq. (1) where \( P \) represents the set of all the hardware devices.

\[
\exists P \in \mathbb{P} \mid F_{\text{maxQL}(P)} \geq T_{\text{hmaxQL}} \land \\
F_{\text{maxHwUtil}(P)} \geq T_{\text{hmaxHwUtil}}
\]
CPS: \{HWnode\} affectExactly (QL, U);
BLOB: \{SWop, DYmsg\} affectAtLeast \{U\};
P&P: \{SWop, DYmsg\} affectAtLeast \{TH, U\};
EP: \{SWop, DYmsg\} affectAtLeast \{RT, U\};
EST: \{DYmsg\} affectAtLeast \{RT, U\};
OLB: \{SWcomp, SWop, DYmsg\} affectAtMost \{RT, WT\};
TR: \{SWop\} affectExactly \{RT, TH\};

\textbf{Listing 3.} Performance Antipattern-based rules to reduce model-to-results uncertainty.

AUTH: \{DYmsg\} affectAtLeast \{SEC, RT, TH\};
RBAC: \{SWop, DYmsg\} affectAtLeast \{SEC, RT, TH\};
LV: \{SWcomp, SWop, DYmsg\} affectAtLeast \{SEC, RT, TH\};
SES: \{HWnode\} affectAtMost \{SEC, U, QL, WT\};

\textbf{Listing 4.} Security Pattern-based rules to reduce model-to-results uncertainty.

CPS is an antipattern that occurs when processes cannot make effective use of available hardware devices to a non-balanced assignment of tasks. The over-utilized hardware devices are detected by checking if the queue length and the utilization overcome pre-defined thresholds.\footnote{A specific characteristic of performance antipatterns is that they contain numerical parameters representing thresholds (e.g., high utilization, excessive number of messages). For further details refer to Cortellessa et al. (2014).}

\subsection{4.2.2. Security patterns}

Security patterns deal with the security properties of software systems. In literature, security patterns have gained significant attention by the research community after the seminal work in \textit{Yoder and Barcalow} (1998) that have defined a catalog of 7 patterns collaborating to provide the necessary security within an application. The benefit of using these patterns is that software systems should result in a ‘more secure’ design, i.e., they should expose fewer security flaws at the design level, and several works recently have demonstrated their usefulness (Yskout et al., 2015; Nguyen et al., 2015; Fernandez-Buglioni, 2013).

\textbf{Listing 4} reports the traceability rules while considering the specification of some security patterns (\textit{Yoder and Barcalow}, 1998), i.e., Authorization (AUTH), Role-based Access Control (RBAC), Limited View (LV), and Session (SES), respectively. These rules come from our understanding of the pattern specification in \textit{Yoder and Barcalow} (1998). In particular, for each pattern we defined the model elements (i.e., SWcomp, SWop, DYmsg, HWnode) that most likely affect their security level (SEC). To further extend their specification we also consider the performance indices related to that specific element (i.e., RT, TH, U, QL, WT).

For example, introducing a Session (SES) pattern indicates that HWnode needs to verify system accesses thus to prevent malicious users, hence it affects at most its security level (i.e., SEC) and all performance indices related to that specific model element (i.e., U, QL, and WT). In all the traceability rules we do not use the affectExactly construct, since we assume that model elements involved in security properties may affect other result elements. In this way we do not restrict model elements in their involvement to further extra-functional properties.

\textbf{Fig. 5} reports the weighted footprint graph, after introducing performance and security (anti)pattern-based rules. For figure readability, it is built considering: (i) CPS and TR performance antipattern-based rules (see \textbf{Listing 3}); (ii) AUTH and SES security pattern-based rules (see \textbf{Listing 4}). The inclusion of these performance antipatterns and security patterns generates nine additional traces (bold lines) and three no-traces (dashed lines) between MES and RES. In fact, the weighted footprint graph from input (see \textbf{Fig. 3}) shows 0 traces (T), 12 no-traces (N), and 18 uncertain traces (TN), whereas the weighted footprint graph, after (anti)pattern-based rules (see \textbf{Fig. 5}), shows 9 traces (T), 15 no-traces (N), and 6 uncertain traces (TN). Note that the newly generated traceability information triggers the modification of weights associated to model elements. \textbf{Fig. 6} shows the weighted footprint graph that is obtained after antipattern-based rules augmented with weight and guilt estimates reported in \textbf{Listing 2}, where weights of RE nodes are modified accordingly. For example, the SWcomp node is here weighted by the following calculation: \(0.5 \times 0.42 + 0.33 \times 1 = 0.54\), in fact this node is involved in the (SWcomp, SEC) trace. Further nodes are weighted: the (SWop, RT) trace leads to assign the weight equal to 0.5 to the SWop node; the (DYmsg, RT) and (DYmsg, SEC) traces produce a weight equal to \(0.5 + 0.33 = 0.83\) for the DYmsg node.

Note that the specification of performance and security (anti)pattern-based rules may also keep unchanged or even increase the overall uncertainty of the system since no logical con-
sequences can be deduced while considering the addition of these further constructs.

5. Validation

Our approach enables the specification of traceability between architectural model elements and extra-functional results. In the following we report the experimental results on three different case studies, i.e., two academic research projects (i.e., ECCO and DesignSpace) and one production system (i.e., PermissionSystem). All the case studies share the same experimental setting, as discussed hereafter.

5.1. Experimental setting

Our approach is implemented in a tool called SoEfTraceAnalyzer (Trubiani et al., 2016) that is built on the Eclipse platform to improve its usability. Fig. 7 shows the screenshot of the tool used to model our DesignSpace illustrative example (see Section 2). It highlights four different input views:

1. the textual input view includes the specification of perspectives along with their constituent elements, and dependencies (i.e., affectAtLeast, affectAtMost, and affectAtMost) among the elements that are regulated with our language for expressing traceability;
2. the outline view lists source (i.e., SW.accessModels, SW.selectModels, and SW.loadModels) and target (i.e., RT.loadModels, and SEC.accessModels) artifacts. Note that target artifacts match the R2 and R3 extra-functional properties reported in Table 1b as performance and security requirements, respectively;
3. the footprint graph view depicts the source and target artifacts in the middle of the graph as nodes, dependencies are reported as arcs where bold lines denote traces and dashed lines indicate no-traces;
4. the trace matrix view reports source and target artifacts; an entry (i, j) of the matrix indicates the relationship between the i-th
source elements vs the jth target element, where labels T, N, TN mean 'trace', 'no-trace', and 'uncertain trace', respectively.

Our experimentation is conducted by using the input provided by engineers, and adding further knowledge about patterns and antipatterns. Specifically, security patterns are explicitly stated in the system specification, whereas detection of performance antipatterns is performed by means of our rule-based engine (Cortellessa et al., 2014). Both security patterns and performance antipatterns are translated with our traceability language and may contribute to reduce uncertainty. As anticipated in Section 4.1.2, two further properties can be defined for this goal, i.e., weight and guilt estimates that allow to quantify violations of extra-functional results and how much architectural elements contribute to requirements’ violations, respectively.

To investigate the effectiveness of traceability links, provided as output by our tool, we conducted the following experimentation. If a performance/security flaw is identified we apply a set of refactoring actions (e.g., substitute a software component with a faster/lower one, re-deploy a certain software component) to quantify the impact of such heuristics on performance/security results. Specifically, without traces we verify the impact of the refactoring by modifying all architectural elements contributing to that performance/security flaw one-by-one. On the contrary, with traces we restrict the refactoring to the subset of model elements indicated by the traceability links. This leads to quantify the effectiveness of our output graph as support to software designers since it provides guidelines to modify specific architectural elements with the goal to fulfill extra-functional requirements.

5.2. Selection of extra-functional properties

As stated in Section 1, our approach is not limited, in principle, to specific extra-functional properties. However, to investigate the effectiveness of traceability links we decided to focus on performance and security since this latter property has a “direct” overhead on the former. Besides this, performance and security trade-offs have been considered in our previous work (Cortellessa et al., 2010; Rodríguez et al., 2012) where we introduced a framework to support the analysis of performance degradation due to the introduction of different security mechanisms. Such framework is able to quantify the system performance degradation while varying the adopted security strategies. Hence, we used this framework to translate (annotated) software architectural models, enriched with security annotations, to obtain a performance model, i.e., a Queueing Network (QN) (Lazowska et al., 1984), that embeds the stated security requirements.

As drawback, QN models only provide performance results, and they do not allow to estimate the achieved security level. In fact, it is very challenging to estimate the security of a software component, service, a hardware device, a firewall, etc. To this end, we leave to engineers the decision of associating architectural model elements with pre-defined security levels, i.e., low, medium, and high. To increase/decrease such levels we expect that the corresponding operation has a larger/lower demand in terms of computational resources and we reflect this change into the performance model.

The performance analysis has been conducted by solving the QN model with two well-assessed techniques (Jain, 1991), i.e., mean value analysis (MVA) and simulation. Both solution techniques are supported by Java Modeling Tools (JMT) (Casale and Serazzi, 2011).

5.3. Case study 1: ECCO

ECCO is an academic research project that aims to reuse existing software and customize it to meet customer-specific needs (Fischer et al., 2014; 2015). The name stands for Extraction and Composition for Clone-and-Own, and it is a framework enabling the automated systematic reuse of existing arbitrary development artifacts. The workflow consists of three main operational steps: (i) extraction that locates and extracts desired reusable artifacts (e.g., code) from existing variants; (ii) composition where the selected artifacts are pasted/edited to create the new product; (iii) completion where the new product variant is adapted to account for needs that did not exist previously.

Fig. 8 reports an excerpt of the ECCO software architectural model. It is constituted by 17 software components (i.e., SC, with \( i \in \{1, \ldots, 17\} \)) among which 2 components have been intentionally added to model the security patterns that are AUTH and RBAC. Software components are labeled with the stereotype artifact since they are deployed on five different hardware machines labeled with the stereotype GaExecHost meaning the execution hosts. Hardware platforms communicate through different networks, i.e., wide and local area networks, labeled with the stereotype GaCommHost that indicates communication hosts.
The system workload is assumed to be closed with a population of 1500 users and an average thinking time of 1 second. The performance requirements that we consider, under the stated workload, are: (i) RT: the average response time of the composition service has to be less than 5 ms; (ii) TH: the throughput of the extraction service has to be greater than 0.3 requests/ms. Security requirements indicate that there is a critical service (i.e., addProduct) that require a high security level (SL).

While solving the QN built for ECCO we found that all the requirements are not fulfilled. In particular, the average response time (RT) of the composition service has been estimated to be 10.09 ms, the throughput of the extraction service has been evaluated to be 0.16 requests/ms. Also the security requirement is not fulfilled since the addProduct service is annotated in the software architectural model with the medium security level.

Performance antipatterns have been detected by means of our rule-based engine (Cortellessa et al., 2014), and we found the following two instances: (i) The Ramp (TR) antipattern, i.e., the response time of the composition service is quite unstable along simulation time; (ii) Pipe & Filter (P&F) antipattern, i.e., the throughput of the extraction service suffers from the consistencyCheck and dependencyResolution slow filters.

Fig. 9 reports an excerpt of the ECCO Weighted Footprint Graph after (anti)pattern-based rules, where we can notice that the response time (RT) of the composition service shows traces with SC1, SC3, and SC6 software components. In particular, the weight reported on the RTcomposition result element represents the observed requirement violation (0.5), i.e., the overhead of the predicted value (10.09 ms) with respect to the requirement (5 ms). The guiltiness values associated to the software components indicate that they contribute to that specific performance flaw at a certain extent, specifically with the following estimates: SC1 (0.6), SC3 (0.26), and SC6 (0.49).

Fig. 10 depicts the response time of the composition service, where the horizontal bold line indicates the performance requirement. The leftmost side of the figure reports the initial system and we can see that the service is provided in 10.09 ms. The middle part of the figure reports the response times observed while refactoring the software components one-by-one, without the support of traces. In this case software components are replaced with ones two times faster to understand at which extent response times may improve. In the figure we can notice that there are few results showing a reasonable improvement whereas all other predicted values are very similar to the initial system. In particular, the refactorings associated to the software components suggested by our weighted footprint graph (see Fig. 9) show a certain improvement for the software system, in fact modifying SC3, SC5 and SC6 we get 6.48 ms, 8.31 ms, and 7.43 ms, respectively. It is worth to notice that guilt estimates are also reflected in the experimental results: SC3 shows the highest guilt value (0.6, see Fig. 9) and its refactoring actually improves the response time of the composition service (from 10.09 ms to 6.48 ms), whereas SC5 shows the lowest guilt value (0.26, see Fig. 9) and the performance improvement is lower (from 10.09 ms to 8.41 ms). The rightmost side of the figure shows the response time obtained while jointly refactoring SC1, SC3, and SC6 software components. This leads to a response time equal to 2.81 ms, thus allowing the fulfillment of the stated requirement.

Fig. 11 depicts the throughput of the extraction service. Similarly to Fig. 10, the horizontal bold line indicates the performance
requirement. The leftmost side of the figure reports the initial system and we can see that 0.16 reqs/ms are served. The middle part of the figure reports the throughputs observed while modifying the software components one-by-one, without the support of traces. Also in this case software components are replaced with one two times faster to understand at which extent throughputs may improve. We can notice that the best improvement makes the system to serve 0.25 reqs/ms. The rightmost side of the figure shows the throughput obtained while using the traces, i.e., the SC4, SC6, SC7, and SC8 software components are jointly refactored. This leads to a throughput equal to 0.33 reqs/ms, thus allowing the fulfillment of the stated requirement.

To increase the security level (SL) of the addProduct service from medium to high we modify the ECCO system by replacing the software component providing the associated security mechanism (i.e., SC18) with one component two times slower to understand at which extent performance may degrade. In this case traceability information is used to refactor all the other software components contributing to that service that are not providing security strategies (i.e., SC2, SC10). These last components are replaced with ones two times faster to quantify their impact on the overall service provisioning.

Table 1 compares the experimental results with and without the traceability information. The first column reports the stated requirements, the second column shows the results from the analysis of the initial system, the third column reports the results without traces, and the fourth column shows the results obtained while using the traceability information. In the last column of the table we report the percentage of improvement gained with traceability links vs without such information. For example, in the first row of the table we can see that the RT of the composition service improves from 10.09 ms to 6.48 ms (without traces) up to 2.81 ms (with traces). Hence, with traceability links, an improvement of 56.6% is achieved. Last row of table reports the security requirement and the refactoring enables to increase the security level from medium to high. In this case the RT of the addProduct service degrades from 2.06 ms to 2.95 ms, hence traceability links lead to get 2.47 ms, i.e., 16.27% of improvement.

5.4. Case study 2: DesignSpace

DesignSpace is an academic research project (Demuth et al., 2015) that has been illustrated in Section 2. It implements two security patterns. First, the Limited View (LV) allows to restrict the visibility of artifacts under development to a restricted number of users belonging to some ad-hoc groups. Second, the Role-based Access Control (RBAC) enables permissions of adding, modifying, and deleting artifacts on the basis of user roles, such as programmers, project managers, etc.

We omit the graphical representation of the software architectural model, however it includes eighteen software components (SC1, ..., SC18) and one hardware device (HW.propertyManagement).

The system workload is assumed to be closed with a population of 1800 users and an average thinking time of 1 s. The performance requirements that we consider, under the stated workload, are: (i) TH: the throughput of the createArtifact service has to be greater than 0.5 requests/ms; (ii) U: the utilization of the HW.propertyManagement hardware device has to be lower than 70%. Security requirements indicate that there is a critical service (i.e., commit) that require a medium security level (SL).
While solving the QN built for the DesignSpace we found that all the performance requirements are not fulfilled. In particular, the average throughput (TH) of the createArtifact service has been estimated to be 0.1 reqs/ms, and the HWpropertyManagement hardware device has an utilization equal to 81%. Also the security requirement is not fulfilled since the commit service is annotated with low security level.

Detection of performance antipatterns provides as output the following three instances: (i) Blob antipattern, i.e., the getWorkspace software component is managing most of the application’s business logics; (ii) Extensive Processing (EP) antipattern, i.e., the demand of the showArtifactDetails service is much larger than createArtifact service; (iii) Empty Semi Trucks (EST) antipattern, i.e., the software components startValidation and showValidation communicate with a large number of messages of low size.

Fig. 12 depicts the throughput of the createArtifact service, where the horizontal bold line indicates the performance requirement. The leftmost side of the figure reports the initial system and we can see that 0.1 reqs/ms are served. The middle part of the figure reports the throughputs observed while modifying the software components one-by-one, without the support of traces. In this case software components are replaced with ones ten times faster to understand at which extent performance may improve. We report all values in the figure and we can notice that the best improvement makes the system to serve 0.38 reqs/ms. The rightmost side of the figure shows the throughput obtained while using the traces, i.e., the SC7, SC10, SC12, and SC13 software components are jointly refactored. This leads to a throughput equal to 0.52 reqs/ms, thus allowing the fulfillment of the stated requirement.

Fig. 13 depicts the utilization of the propertyManagement hardware device, where the horizontal bold line indicates the performance requirement. The leftmost side of the figure reports the initial system and we can see that the utilization is 0.81%. The middle part of the figure reports the utilization values while modifying the software components one-by-one, without the support of traces. In this case software components are replaced with ones ten times faster to understand at which extent utilization may improve. We can see in the figure that the best value is achieved while refactoring the SC7 component that leads to an utilization of 0.71%. The rightmost side of the figure shows the utilization obtained while using the traces, i.e., the SC7 and SC10 software components are jointly refactored. This leads to an utilization equal to 0.61%, thus allowing the fulfillment of the stated requirement.

To increase the security level (SL) of the commit service from low to medium we modify the DesignSpace system by replacing the software component providing the associated security mechanism (i.e., SC4) with one two times slower to understand at which extent performance may degrade. In this case traceability information is used to refactor all the other software components contributing to that service that are not providing security strategies (i.e., SC5, SC7, SC8). These last components are replaced with ones two times faster to quantify their impact on the overall service provisioning.

Similarly to Table 1, Table 2 summarizes the experimental results with and without the traceability information. For example, in the first row of the table we can see that the TH of the createArtifact service improves from 0.1 reqs/ms to 0.38 reqs/ms (without traces) up to 0.52 reqs/ms (with traces). Last row of table reports the security requirement and the refactoring increases the security level from low to medium. In this case the RT of the commit service inevitably degrades, in particular it goes from 0.82 ms to 1.63 ms. Traceability information leads to get 1.46 ms, i.e., 9.81% of improvement. We also report the percentage of improvement while comparing the results with and without the traceability information, and we can notice an improvement up to 26.92% among all requirements.

5.5. Case study 3: PermissionSystem

PermissionSystem is an industrial system developed to manage security permissions in a content management system. It is designed to meet multiple security patterns on content and users at the same time. In its actual version the PermissionSystem implements three security patterns. First, it provides direct permissions (authorization pattern) assigned to content entities and/or users. For instance the administrator of a software product may mark an
entity as visible for all users. In such case the entity will be visible for every user regardless of his/her account in the system. The administrator may also assign the permission to a user account, allowing it to view all entities under a given structure. Second, the PermissionSystem implements the session security pattern which is mostly needed to store permissions assigned to user accounts at runtime. This is also a performance optimization as it allows for faster evaluation of user permissions. Third, a role based access control (RBAC) pattern is implemented to support more sophisticated permissions. In this pattern, a mapping between user accounts and their respective permissions are created. This feature allows the integration of PermissionSystem with other heterogeneous systems (e.g., active directory).

We omit the graphical representation of the software architectural model, however it includes eight software components ($SC_1, \ldots, SC_8$) and two hardware devices ($HW.permissions, HW.accesses$).

The system workload is assumed to be closed with a population of 80 users and an average thinking time of 1 s. The performance requirements that we consider, under the stated workload, are: (i) $RT$: the average response time of the permissionOnEntities service has to be less than 5 ms; (ii) $U$: the utilization of the $HW.permissions$ hardware device has to be lower than 90%. Security requirements indicate that there are some critical services (i.e., accessPermission and checkPermission) that require a high security level (SL).

While solving the QN built for the PermissionSystem we found that all the performance requirements are not fulfilled. In particular, the average response time ($RT$) of the permissionOnEntities service has been estimated to be 15.72 ms, and the $HW.permissions$ hardware device has an utilization equal to 100%. Also the security requirements are not fulfilled since the checkPermission is annotated with medium security level.

Detection of performance antipatterns provides as output the following two instances: (i) Concurrent Processing System (CPS) antipattern, i.e., $HW.permissions$ hardware device is over-utilized; (ii) One-Lane Bridge (OLB) antipattern, i.e., the response time of the permissionOnEntities service suffers from the low multithreading enabled in the database while verifying permissions.

Fig. 14 depicts the response time of the permissionOnEntities service, where the horizontal bold line indicates the performance requirement. The leftmost side of the figure reports the initial system and we can see that the service is provided in 15.72 ms. The middle part of the figure reports the $RT$ observed while modifying the software components one-by-one, without the support of traces. In this case software components are replaced with ones ten times faster to understand at which extent performance may improve. The best value is obtained while refactoring the $SC_4$ software component, in fact the response time slows down to 7.06 ms. The rightmost side of the figure shows the response time obtained while using the traces, i.e., the $SC_4$ and $SC_5$ software components are jointly refactored. This leads to a response time equal to 4.98 ms, thus allowing the fulfillment of the stated requirement.

Fig. 15 depicts the utilization of the two hardware devices (i.e., $HW.permissions$, $HW.accesses$), where the horizontal bold line indicates the performance requirement. The leftmost side of the figure reports the initial system and we can see that the utilization of $HW.permissions$ device is 100%, whereas the utilization of $HW.accesses$ device is 39%. The middle part of the figure reports the utilization values while redeveloping the software components one-by-one, without the support of traces. We can see in the figure that the best value is achieved while moving the $SC_4$ component that leads to an utilization of 26% for the $HW.accesses$ device. The rightmost side of the figure shows the utilization obtained while using the traces, i.e., the $SC_2$, $SC_3$ and $SC_4$ software components are jointly redeveloped from $HW.permissions$ to $HW.accesses$. This leads to an utilization equal to 81% for $HW.permissions$ device, but the hosting node shows an increasing utilization (from 39% to 84%). However, both devices fulfill the stated requirement.

To increase the security level (SL) of the checkPermission service from medium to high we modify the PermissionSystem by replacing the software component providing the associated security mechanism (i.e., $SC_7$) with one ten times slower to understand at which extent performance may degrade. In this case traceability information is used to refactor the software components contributing to that service that are not providing security strategies (i.e., $SC_1$, $SC_2$, $SC_3$). These last components are replaced with ones ten times faster to quantify their impact on the overall service provisioning.

Similarly to Table 2, Table 3 summarizes the experimental results with and without the traceability information. For example, in the first row of the table we can see that the $RT$ of the permissionOnEntities service improves from 15.72 ms to 7.06 ms (without

<table>
<thead>
<tr>
<th>Extra-functional requirements</th>
<th>Performance analysis</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TH(SW.createArtifact) \geq 0.5 \text{ reqs/ms}$</td>
<td>0.1 reqs/ms</td>
<td>0.38 reqs/ms</td>
</tr>
<tr>
<td>$U(HW.propertyManagement) \leq 0.7$</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>$SL(commit) = \text{medium}$</td>
<td>0.82 ms</td>
<td>1.63 ms</td>
</tr>
</tbody>
</table>

---

**Table 2** Performance improvement gained with the model-to-results traces in the DesignSpace case study.
traces) up to 4.98 ms (with traces), i.e., 29.49% of improvement. Last row of table reports the security requirement and the refactoring enables the security level to increase from medium to high. In this case we report the RT of the checkPermission service that degrades from 1.86 ms to 3.45 ms. Traceability information indicates refactorings leading to 3.05 ms, i.e., 11.59% of improvement. While comparing the results with and without the traceability information, we can observe an improvement up to 29.46% among the three requirements.

5.6. Users feedback

To investigate the usefulness of our approach, we conducted an experiment with users to collect their feedback, with the goal to answer the following two research questions:

RQ1: Do trace uncertainties for architectural models vs extra-functional results exist and are relevant in practice?

RQ2: Do security patterns and performance antipatterns help in the reduction of uncertainty?

Our experiment was performed by contacting the 3 owners of the presented case studies, plus 9 PhD students involved in the Computer Science program at the Gran Sasso Science Institute (CSSI), Italy. All the PhD students had been exposed to software architecture modeling and extra-functional analysis topics in their courses. Hence, the subjects involved in the experiment were 12 users aware of the domain.

The outcome of our user study is that: 100% of the subjects confirmed that there are uncertainties while bridging architectural model elements with extra-functional results by answering the RQ1 question positively, and this high percentage confirms our claim that uncertainties exist and are relevant in practice; 83% of the subjects expressed a positive opinion on the addition of patterns and antipatterns knowledge, however 2 users over 12 were skeptical for the RQ2 question, and they provided the following motivations:

- patterns and antipatterns are application-dependent, i.e., their usefulness might be strongly related to the case study under analysis, hence they might not lead to a better interpretation of results in some cases;
- patterns and antipatterns are granularity-related, i.e., their usefulness is affected by the granularity used in the specification of architectural model elements and extra-functional results, hence if they are expressed on different levels of granularity they become useless.

We agree with these motivations and we think that, even if the usefulness of patterns and antipatterns cannot be always guaranteed, in our case studies their usage resulted to provide a consistent uncertainty reduction. To further investigate this point, we asked all users to provide their own traceability links for the three case studies under analysis, and we quantified their outputs by evaluating the average percentage of uncertainty reduction (see Table 4). Further discussion on this experimentation is reported in Section 5.7.

Besides this, we also asked for further comments on the approach and on the used tool, thus to understand the difficulties of users in the assigned tasks. One user pointed out that it could be useful to identify architectural elements and extra functional properties that share the same names and automatically produce their traceability links, leaving the opportunity to the user to modify or delete such links afterwards. Another user complained about the typing of the uncertainties, it may take too much time and it may be an error-prone task, i.e., a smarter code completion and a GUI could be helpful. Both these points represent our future work.

5.7. Afterthoughts

The experimentation conducted on three different case studies supports the idea of exploiting traceability information in the field of extra-functional analysis. We showed that with traces we are able to get a certain performance improvement across all the presented case studies. Specifically, our approach allows to derive quantitative evaluations of performance and security trade-offs, and it is suitable for decision making while refactoring software architectures. As feedback to software designers we are able to identify the root causes of performance/security flaws and to point out the most promising model elements for the refactoring process by means of traceability links.

Table 4 provides an overview of the presented case studies, where rows report some system features, and columns list the three analyzed systems: two academic research projects (i.e., ECCO and DesignSpace) and one production system (i.e., PermissionSystem). For each considered case study, in the upper part of the ta-

---

**Table 3**

<table>
<thead>
<tr>
<th>Extra-functional requirements</th>
<th>Performance analysis</th>
<th>Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT(\text{permissionOnEntities}) ≤ 5 ms</td>
<td>15.72 ms</td>
<td>7.06 ms</td>
</tr>
<tr>
<td>U(\text{HW permissions}) ≤ 0.9</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>SL(checkPermission) = high</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>RT(checkPermission)</td>
<td>1.86 ms</td>
<td>3.45 ms</td>
</tr>
</tbody>
</table>
ble we report: (i) the number of architectural model elements; (ii) the number of extra-functional properties; (iii) the size of trace matrix that basically represents \#rows * \#columns entries. In the lower part of the table we report: (i) detected performance antipatterns; (ii) specified security patterns; (iii) the uncertainty reduction is quantified in terms of number of traces (T), no-traces (N), and their sum (T+N) that we have in the transition matrix before and after (anti)patterns knowledge. Uncertainty reduction is reported by evaluating the traceability links provided by the owners of the three case studies, and also calculating the average of the traceability links specified by all the other participants involved in our user study. These average estimations inevitably result in decimal numbers for both the number of traces and no-traces.

For example, in the last column of the table, we can see that for the PermissionSystem we have 16 architectural model elements, 58 extra-functional properties obtaining a matrix 16 * 58 = 928 entries. While adding performance antipatterns (i.e., CPS and OLB) and security patterns (i.e., AUTH, SES, and RBAC) traces from the owner of the case study go from 27 to 44, whereas no-traces go from 252 to 420. Both information allowed to reduce the initial uncertainty (928 – 279 = 649 TN) of a considerable amount (928 – 464 = 464 TN), i.e., 28.5% of uncertainty reduction. From the other participants we found that traces go from an average value of 24.4 to 34.1, whereas no-traces go from an average value of 46.3 to 208.8. Both information allowed to reduce the initial uncertainty (928 – 70.7 = 857.3 TN) of a substantial amount (928 – 242.9 = 685.1 TN), i.e., 20.1% of uncertainty reduction. It is worth to remark that even if the absolute values of traces and no-traces from the owners is higher than the ones provided by the other participants, our experimentation provides evidence that (anti)patterns are helpful for the uncertainty reduction.

As stated in Section 4.2, the specification of performance and security (anti)pattern-based rules does not guarantee to decrease the overall uncertainty of the system. In fact, in Table 4 we can notice that the DesignSpace case study shows an unchanged number of no-traces (i.e., 1941), whereas traces rather increase (from 72 to 87) while considering the specification of traceability links from the owners of the case study. Interestingly, looking at the average values from the other participants we also found an unchanged number of no-traces (i.e., 57.9). Even if the absolute value is much lower in this last case, the experimentation shows that (anti)patterns were not useful for the understanding of no-traces in this case study. In fact, for the DesignSpace case study the initial uncertainty (2820 – 2013 = 807 TN) is reduced of a small amount (2820 – 2028 = 792 TN), i.e., 1.9% of uncertainty reduction. Indeed uncertainty reduction strongly depends from the initial input provided by software engineers, however in our experimenta-

<table>
<thead>
<tr>
<th></th>
<th>ECCO</th>
<th>DesignSpace</th>
<th>PermissionSystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArchModel</td>
<td>43</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>Elements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExtraFunc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Properties</td>
<td>70</td>
<td>94</td>
<td>58</td>
</tr>
<tr>
<td>Size of TM</td>
<td>3010</td>
<td>2820</td>
<td>928</td>
</tr>
<tr>
<td>Performance</td>
<td>TR</td>
<td>BLOB</td>
<td>CPS</td>
</tr>
<tr>
<td>Antipatterns</td>
<td>P&amp;R</td>
<td>EP, EST</td>
<td>OLB</td>
</tr>
<tr>
<td>Security</td>
<td>AUTH</td>
<td>LV</td>
<td>AUTH</td>
</tr>
<tr>
<td>Patterns</td>
<td>RBAC</td>
<td>RBAC</td>
<td>SES, RBAC</td>
</tr>
<tr>
<td>Uncertainty reduction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Owners</td>
<td>T: 48 → 60</td>
<td>T: 72 → 87</td>
<td>T: 27 → 44</td>
</tr>
<tr>
<td></td>
<td>N: 713 → 781</td>
<td>N: 1941 → 1941</td>
<td>N: 252 → 420</td>
</tr>
<tr>
<td></td>
<td>T+N: 761 → 841</td>
<td>T+N: 2013 → 2028</td>
<td>T+N: 279 → 464</td>
</tr>
<tr>
<td>Other</td>
<td>T: 29.8 → 36.8</td>
<td>T: 218 → 31.1</td>
<td>T: 24.4 → 34.1</td>
</tr>
<tr>
<td></td>
<td>N: 133.9 → 185.0</td>
<td>N: 57.9 → 57.9</td>
<td>N: 46.3 → 208.8</td>
</tr>
<tr>
<td></td>
<td>T+N: 163.7 → 221.8</td>
<td>T+N: 79.7 → 89</td>
<td>T+N: 70.7 → 242.9</td>
</tr>
</tbody>
</table>

Table 4: Overview of the presented case studies.
antipatterns) hence extra-functional improvements are not guaranteed.

6. Threats to validity

Besides inheriting all limitations of the underlying software quality engineering and model-driven traceability techniques (Kan, 2002; Gotel et al., 2012), our approach exhibits the following threats to validity.

Correctness: input is given by the engineer that defines uncertainty constructs to the level of detail she or he is comfortable with. This means that not every input combination is valid and it becomes increasingly unlikely that the input remains consistent, especially if the input is provided by different engineers. An important aspect of future work is to provide correctness checks based on the consistency of the input, in fact consistency does not imply correctness. We can identify the input that is responsible for incorrectness and granularity problems, and provide support to engineers for resolving the detected issues.

Granularity: it is difficult to establish at what level of granularity traces between model and results should be generated. Extra-functional properties can be estimated at different levels of granularity, e.g. the response time performance index or the security level can be evaluated at the level of a cpu device, or at the level of a service that spans on different devices. Then, the engineer has the choice to establish traceability between the model elements and it is unrealistic to keep under control all performance and security properties at all levels of abstraction.

Selection of extra-functional properties: our current experimentation is focused on performance and security flaws, but many other extra-functional properties can be tackled as well. The rationale of selecting these two is that there exists a trade-off relationship among them: the raising of the security level implies a performance degradation that can be mitigated with model-to-results traces. In the near future we plan to integrate larger set of extra-functional properties, such as reliability, along with other methods, such as fault-tree analysis (Lyu et al., 1996), to derive further extra-functional results (e.g., mean time to failure, mean time to repair) and deeply investigate the traceability benefits in this domain.

Complexity of architectural refactorings: to quantify the effectiveness of model-to-results traces we applied a set of refactoring actions (e.g., substitute a software component with another one that is x times faster/lower, re-deploy a certain software component) to the subset of model elements identified by traceability links. These refactorings are used for arguments sake to explore trade-offs and to demonstrate the usefulness of the weighted footprint graph, in fact we found that highly weighted architectural elements are the ones that actually contribute to higher performance benefits. However, such refactorings do not guarantee performance improvements a priori, because the entire process is based on heuristic evaluations. In fact, refactoring of architectural models is a very complex process (Mens and Tourné, 2004), and it may also include the addition and deletion of other model elements that affect our footprint graph in its structure. This last point opens an interesting direction of research that we aim to investigate in the near future.

Complexity of security properties: it is very challenging to target security concerns because different types of attacks may occur and a single compromise can cause system failure. For sake of illustration, in our experimentation we considered pre-defined security levels (i.e., low, medium, and high), and the increase/decrease of such levels is regulated by larger/lower demand of computational resources, thus to explore security vs performance trade-offs. However, we plan to enable interoperability among extra-functional results, e.g., the occurrence of security flaws in architectural elements may lead to reset the performance results traced by such elements, thus to strengthen the security-related issues and explicitly include system reliability in the set of considered extra-functional results.

Specification of patterns and antipatterns: to reduce model-to-results uncertainty we use specific structures for the specification of performance antipatterns and security patterns, and these structures strictly conform to our interpretation of the literature (Yoder and Barcalow, 1998; Smith and Williams, 2012). Indeed several other feasible interpretations of (anti)patterns can be provided due to their wide variety of implementation variants, which can lead to higher/lower accuracy. This unavoidable gap is an open issue, and requires a wider investigation to consolidate the (anti)patterns’ specification in this domain.

Sufficiency of patterns and antipatterns: to build model-to-results traces we exploit performance antipatterns and security patterns since they are recognized to provide solid solutions to known and recurrent problems (Kocing, 1998). However, due to the complexity of real world systems their usage may not be sufficient for the detection of extra-functional problems. As future work, we plan to investigate further methodologies (Jin et al., 2012; Chen and Wagner, 2002) to catch extra-functional issues and integrate their output in our specification.

Uncertainty reduction: the addition of security patterns and performance antipatterns does not guarantee the uncertainty reduction a priori. As future work, we plan to integrate other approaches to derive model-to-results traceability links. For example, to achieve performance goals we can consider bottleneck analysis (Franks et al., 2006) and model optimization methods (Aleti et al., 2013) to improve the uncertainty reduction.

Simulation accuracy: in our experimentation performance analysis results are obtained by simulating the QN model with JMT (Casale and Serazzi, 2011). In particular, JMT performs an automatic step of the simulation when all performance metrics are estimated with the required accuracy. We used the default values for the Confidence Interval Size of the solution and for the Max Relative Error of the greatest sample error that are 0.99 and 0.03, respectively, but they can be modified if more/less accurate results are required. Even if in literature some approaches have been proposed to combine QN prediction results with empirically found data to get more accurate QN-based models (Liu et al., 2005), for our case studies (see Section 5) the numerical results are used to compare architectural refactorings applied to different model elements to validate if traceability links point out the most critical ones. Hence, we think that the considered simulation results cannot heavily affect the validation process, but we intend to investigate this aspect in the near future.

7. Related work

The work presented in this paper relates to two main research areas and builds upon our previous results in these areas: (i) software quality engineering, and (ii) model-driven traceability. We also discuss the state-of-art approaches in the context of uncertainty analysis and traceability research dealing with extra-functional properties of software systems.

7.1. Software quality engineering

Software quality represents the entire collection of engineering activities and related analyses used throughout the software development cycle, which are directed to meet extra-functional requirements (Kan, 2002). In literature there are several approaches, surveyed in Dobrica and Niemelä (2002); Aleti et al. (2013), that tackle the problem of analyzing and optimizing software quality at the architectural level. Two main streams of research can be defined in this direction: (i) qualitative approaches provide exploratory insights into the problem, e.g., several analysis methods
have been proposed in Kazman et al. (1994, 1996, 1998) and Tekinerdogan (2004); (ii) quantitative approaches aim to generate numerical data to quantify architectural decisions, e.g., different design alternatives are evaluated in Kazman et al. (2001), Zhu et al. (2005), Grunske (2006) and Martens et al. (2009). Our approach belongs to quantitative approaches, in fact it aims to identify the contribution of architectural elements to extra-functional quantitative results. In literature few related works can be found dealing with the interpretation of extra-functional results, and most of them are based on monitoring techniques and therefore conceived to only act after software deployment. We are instead interested in model-based approaches that can be applied early in the software life-cycle to support design decisions. In this direction we identified two main categories of approaches: (i) rule-based methodologies that define a set of rules to overcome extra-functional problems (Barber et al., 2002; Xu, 2012); (ii) search-based approaches that explore the problem space by examining options to deal with extra-functional flaws (Ipek et al., 2008; Martens et al., 2010). To achieve performance goals, our previous work focused on antipatterns (Smith and Williams, 2003) that are very complex (as compared to other software patterns) because they are founded on different characteristics of software systems related to architectural model elements (e.g., usage dependencies, excessive message traffic) as well as to performance results (e.g., high, low utilization). Performance antipatterns represent effective instruments to tackle the issue of interpreting performance results, because they document: (i) common bad practices leading to performance problems and (ii) solutions in terms of architectural refactoring. Their effectiveness has been demonstrated, among others, by our recently consolidated results: (i) we formalized the representation of antipatterns by means of first-order logic rules that express a set of system properties under which an antipattern occurs (Cortellessa et al., 2014); (ii) we introduced a methodology to prioritize the detected antipatterns and solve the most promising ones (Trubiani et al., 2014a). To deal with security concerns, in our previous work (Cortellessa and Trubiani, 2008; Cortellessa et al., 2010; Rodríguez et al., 2012) we developed a framework to support the specification of security strategies at the architectural level and their translation into QN performance models, thus to estimate security vs performance trade-offs. Moreover, in Cortellessa et al. (2010) we also demonstrated that QN prediction errors never exceed 4% when compared with measurements on the real implementation of the system, thus to assess the accuracy of QN prediction results.

7.2. Model-driven traceability

Traceability has been defined as the ability to describe and follow the life cycle of software artifacts (Lago et al., 2009). The current state of research and practice of traceability in requirements engineering and software development is discussed in Winkler and Pilgrim (2010), where model-driven techniques have been recognized of key relevance to support traceability concerns. In literature, there are many techniques exploiting the automatic recovery of different types of trace links, such as code and models (Antonioli, 2001; Egyed and Grunbacher, 2002), code and documentation (Marcus and Maletic, 2003), code and features (Dagenais et al., 2007). Results from existing empirical studies (Von Knethen, 2002; Briand et al., 2014; Mäder and Egyed, 2015) demonstrate that traceability information has beneficial effects on the effectiveness and efficiency of understanding changes, performing requirements inspections, and evolving software artifacts. In our previous work (Ghabi and Egyed, 2012) we introduced a language for expressing uncertainties in traceability relationships between models and code, which is the main benefit of this technique compared with other traceability approaches. Our recent work (Ghabi and Egyed, 2015) out-passes these techniques by introducing a flexible methodology to express uncertainties, in fact we proved that the same uncertainty expressions could be applied to trace arbitrary kinds of software artifacts. Our approach makes use of this last achievement, in fact here we consider artifacts of different nature that are architectural elements and extra-functional results. This leads to broaden the scope of our research since we introduce a weighting methodology that gives a certain semantics to the involved artifacts: architectural elements are weighted on the basis of how much they contribute to the violation of extra-functional requirements.

7.3. Uncertainty analysis

The problem of dealing with uncertainty in early requirements and architectural decisions has been recognized by several works in literature. In Perez-Palacin and Miranda (2014) a taxonomy for classifying different types of uncertainties is provided, and three main dimensions are outlined: location, level and nature. The location of uncertainty refers to the place where the uncertainty manifests itself within the software artifacts; in this paper it consists of the relationships among architectural elements and extra-functional results. The level of uncertainty is the spectrum of its manifestation between deterministic knowledge and total ignorance; in this paper it is supported by our language and users specify the uncertainty at the level they are comfortable. The nature of uncertainty refers to whether it is due to the imperfection of the acquired knowledge or to the inherent variability of the phenomena being described; in this paper we consider the first nature of uncertainty, but we aim to consider the second type as future work. In Whittle et al. (2009) a language (i.e., RELAX) has been proposed to explicitly address uncertainty for specifying the behavior of dynamically adaptive systems. Our approach differs from this work in the location and nature of uncertainty since Whittle et al. (2009) considers uncertain the system behavior due to inherent variability of the execution environment. In Esfahani et al. (2013) a tool (i.e., GuideArch) has been presented to guide the exploration of the architectural solution space under uncertainty. Our approach differs from this work in the location and level of uncertainty since Esfahani et al. (2013) employs fuzzy mathematical methods to quantify uncertain architectural alternatives, but no levels can be explicitly specified to express higher/lower knowledge on such alternatives. In Letier et al. (2014) a tool (i.e., Moda) has been introduced for multi-objective decision analysis by means of Monte-Carlo simulation and Pareto-based optimization methods. Our approach differs from this work in the location and level of uncertainty since Letier et al. (2014) models uncertainty in parameters’ values as probability distributions that contribute to problem-specific goals, and the level of uncertainty is a discrete value that represents the attainment for each goal and its impact on risk. In our previous work (Ettxebrria et al., 2014; Trubiani et al., 2013) we considered uncertainty in a different location (i.e., performance model parameters), and the analysis relied on lower/upper bounds provided by software architects. Besides the differences in the dimensions of uncertainty, all these works (Whittle et al., 2009; Esfahani et al., 2013; Letier et al., 2014; Ettxebrria et al., 2014; Trubiani et al., 2013) do not explicitly consider extra-functional properties and their traceability with software architectural elements.

7.4. Traceability vs extra-functional properties

In literature there are some approaches that work towards the specification of traceability links between model elements and extra-functional properties. In Fritzsch et al. (2008) a mechanism to annotate performance analysis results back into the original performance models (provided by the domain experts) is presented. On the contrary, our approach includes the software mod-
els for traceability, and it supports the interpretation of analysis results by providing weights on the basis of requirements’ violation. In Petriu et al. (2003) traceability links are maintained between performance requirements and Use Case Map (UCM) scenarios, however these links are used to build Layered Queueing Network (LQN) models only. On the contrary, our approach maps architectural elements with analysis results, thus to support software architects in the task of interpreting these results. In Alhaj and Petriu (2013) traceability links are used to propagate the results of the performance model back to the original software model, however it applies to UML and LQN models only. In contrast to Alhaj and Petriu (2013), our approach aims to automatically build model-to-results traceability links to point out the architectural elements affecting the stated requirements, with no reference to specific modeling languages. In Yu et al. (2008) traceability links are maintained between the software design (i.e., UMLsec models Jurjens, 2001) and the implementation code (i.e., Java) to develop secure and dependable software systems. In particular, the traceability technique proposed in Yu et al. (2008) is intended to find vulnerability bugs in the implementation of security protocols like SSL. On the contrary, our approach is aimed at tracing back security flaws to model elements and applying refactoring at the architectural level. In Bauer et al. (2010) authors extended their work by performing a run-time verification of traceability links from the design model to its implementation, i.e., the actual system is verified against the model while it executes. Our approach can be extended in a similar direction by considering extra-functional results as set of values obtained while changing run-time conditions. In Nhlabatsi et al. (2014) authors present an approach for adaptive information security in the cloud by using traceability as a mean to reason about the relationship between security requirements and the policies that satisfy those requirements. Traceability links are enriched with contextual information in order to support the adaptation of cloud applications to the environment they operate that was unknown or incomplete at design-time. As opposite to Nhlabatsi et al. (2014), our approach builds traceability links to make evident the architectural elements responsible for extra-functional flaws.

Summarizing, we found that some approaches (Fritzche et al., 2008; Petriu et al., 2003; Alhaj and Petriu, 2013) apply on early model abstractions like we do in our approach, whereas the remaining ones (Yu et al., 2008; Bauer et al., 2010; Nhlabatsi et al., 2014) work late in the development process (i.e., at the code level). Our motivation for dealing with early model abstractions is that extra-functional flaws are cheaper to fix (Baniassad et al., 2006), however as drawback the amount of information is limited and an obvious trade-off exists vs late analysis where the results are much more accurate but several constraints have been imposed on the structural and behavioral aspects of a software system. Concluding, to the best of our knowledge, there are no other works in literature that provided a tool-supported approach to derive weights for software artifacts on the basis of traceability uncertainties vs multiple extra-functional results, thus to support software engineers in their interpretation.

8. Conclusion

This paper presents a new approach to automate the traceability between architectural model elements and extra-functional results, thus to support software architects in the identification of the causes that most likely contribute to the violation of extra-functional requirements. To this end, we developed a tool (SoEl-TraceAnalyzer) that is able to interpret a language capable of interpreting uncertainties while capturing model-to-results traceability links. The approach is validated by means of three different case studies, i.e., two academic research projects and one industrial system. The implemented tool and the case studies are publicly available for researchers and practitioners at (Trubiani et al., 2016).

The benefit of the tool is that it allows to automatically visualize the dependencies between modeling elements in architectural models (e.g., software components) and extra-functional properties (e.g., security level, response time, throughput, and utilization). As input the tool takes on the one hand possible influences already known to the domain expert, and on the other hand security patterns and performance antipatterns which express further such dependencies. The specification of extra-functional (anti)patterns is used to make the domain expert dependencies more precise, e.g., by ruling out certain influences.

We conducted a user study and we found that 100% of involved subjects confirmed the relevance of specifying uncertainties while bridging architectural model elements with extra-functional results, and 83% expressed a positive opinion on the addition of patterns and antipatterns knowledge. However, as future work we intend to further investigate the usability of our approach by exposing the developed tool Trubiani et al. to users with higher levels of experience, such as experienced software architects and quality analysts. This wider experimentation will allow us to deeply investigate the usefulness of traceability links for developers in the context of mapping architectural model elements with extra-functional results.

Acknowledgment

We would like to thank the anonymous reviewers for constructive feedback that helped us to improve the paper and all participants of our user study.

References


Vlissides, J., Helm, R., Johnson, R., Gamma, E., 1995. Design Patterns: Elements of Reusable Object-Oriented Software, 49(120). Reading: Addison-Wesley, p. 11.


