Exploiting Traceability Uncertainty Between Software Architectural Models and Performance Analysis Results

Catia Trubiani[1(✉)], Achraf Ghabi[2], and Alexander Egyed[2]

[1] Gran Sasso Science Institute, L'Aquila, Italy
catia.trubiani@gssi.infn.it
[2] Johannes Kepler University, Linz, Austria
a@ghabi.net, alexander.egyed@jku.at

Abstract. While software architecture performance analysis is a well-studied field, it is less understood how the analysis results (i.e., mean values, variances, and/or probability distributions) trace back to the architectural model elements (i.e., software components, interactions among components, deployment nodes). Yet, understanding this traceability is critical for understanding the analysis result in context of the architecture. The goal of this paper is to automate the traceability between software architectural models and performance analysis results by investigating the uncertainty while bridging these two domains. Our approach makes use of performance antipatterns to deduce the logical consequences between the architectural elements and analysis results and automatically build a graph of traces to identify the most critical causes of performance flaws. We developed a tool that jointly considers SOftware and PErformance concepts (SoPeTraceAnalyzer), and it automatically builds model-to-results traceability links. The benefit of the tool is illustrated by means of a case study in the e-health domain.

Keywords: Traceability · Uncertainty · Software modelling · Performance analysis

1 Introduction

In the software development domain there is a very high interest in the early validation of performance requirements because this ability avoids late and expensive repairs to consolidated software artifacts [1]. One of the proper ways to manage software performance is to systematically predict the performance 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 performance objectives [2].

Advanced Model-Driven Engineering (MDE) techniques have successfully been used in the last few years to introduce automation in software performance modeling and analysis [3]. Nevertheless, the problem of interpreting the

D. Weyns et al. (Eds.): ECSA 2015, LNCS 9278, pp. 305–321, 2015.
DOI: 10.1007/978-3-319-23727-5_26

performance analysis results is still quite critical. A large gap exists between the representation of performance analysis results and the software architectural model provided by the engineers. In fact, the former usually contains numbers (e.g., mean response time, throughput variance, etc.), whereas the latter embeds architectural choices (e.g., software components, interaction among components, deployment nodes). Today, such activities are exclusively based on the analysts' experience and therefore their effectiveness often suffers from lack of automation.

In [4] 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 other pieces of code also implement this architectural element. This paper adapts this language to provide model-to-results traceability links while considering typical uncertainties from the performance analysis domain. We presume that engineers know when a given performance result is affected by an architectural element. However, they may not know whether this performance result is also affected by other architectural elements or whether other performance results are also affected by this architectural element.

The knowledge of the engineer is interwoven with software performance antipatterns [5] that represent bad practices in architectural models negatively affecting performance indices. A performance antipattern definition includes the description of a bad practice occurring in the architectural model (e.g., a software component sending an excessive number of messages), along with the solution that can be applied to avoid negative consequences (e.g., high network utilization). In previous work [6] we provided a more formal representation of performance antipatterns by introducing first-order logic rules that express a set of system properties under which an antipattern occurs. The benefit of this representation is that it already includes architectural elements (e.g., software components) and performance results (e.g., utilization) hence it can be used to make the knowledge of engineers less uncertain.

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 performance requirements by jointly considering knowledge from engineers and performance antipatterns. To this end, we developed a tool, namely SoPeTraceAnalyzer [7], that jointly considers SOftware and PErformance concepts: it takes as input a set of statements specifying the relationships between software elements and performance results, and provides as output model-to-results traceability links. The language defined in [4] is extended by adding a weighting methodology that quantifies the performance requirements' violation, thus to highlight the criticality of model elements despite performance results. The key feature of our tool is that the knowledge of performance 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.

The paper is organized as follows: Section 2 presents related work; Section 3 describes our approach; Section 4 illustrates the case study; Section 5 discusses the threats to validity of the approach; Section 6 concludes the paper and outlines future research directions.

2 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 performance engineering (SPE), and (ii) model-driven traceability.

Software performance engineering. SPE represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements [8]. Performance antipatterns [5] are very complex (as compared to other software patterns) because they are founded on different characteristics of software systems, spanning from static through behavioral to deployment. Antipatterns include features related to architectural model elements (e.g., *many* usage dependencies, *excessive* message traffic) as well as to performance results (e.g., *high*, *low* utilization). Our logic-based formalization [6] has been experimented to benefit across different modelling languages [9–11].

Model-driven traceability. In [4] 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. There are many other techniques exploiting the automatic recovery of different types of trace links [12] [13] [14]. Our work [4] out-passes these techniques by introducing a flexible methodology to express uncertainties. We proved in our recent work [15] that the same uncertainty expressions could be applied to trace arbitrary kinds of software artifacts.

In literature there are some approaches that work towards the specification of traceability links between model elements and performance results.

In [16] 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 models for traceability, and it supports the interpretation of analysis results by providing weights on the basis of requirements' violation. In [17] 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. In [18] 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. Our approach instead aims to automatically build model-to-results traceability links to point out the architectural elements affecting the stated requirements.

The problem of dealing with uncertainty in early requirements and architectural decisions has been recognized by several works in literature. In [19] a language (i.e., RELAX) has been proposed to explicitly address uncertainty for specifying the behaviour of dynamically adaptive systems. In [20] a tool

(i.e., GuideArch) has been presented to guide the exploration of the architectural solution space under uncertainty. In [21] a tool (i.e., Moda) has been introduced for multi-objective decision analysis by means of Monte-Carlo simulation and Pareto-based optimisation methods. However, all these works [19–21] do not explicitly consider performance analysis results and their traceability with software architectural elements.

3 Our Approach

Figure 1 illustrates the process we envisage to automate the traceability between architectural model elements and performance analysis results. Ovals in the figure represent operational steps whereas square boxes represent input/output data. Dashed vertical lines divide the process in four different phases.

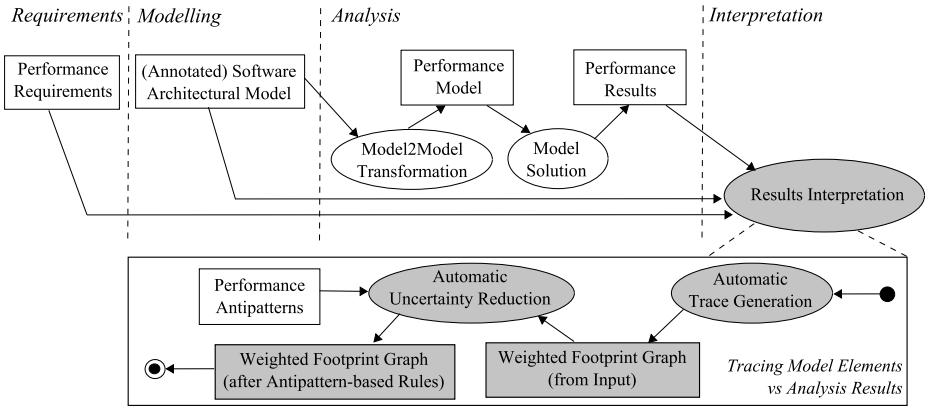


Fig. 1. Deriving automatically model-to-results traceability links by means of performance antipatterns.

We assume that a set of performance *requirements*, among others, is defined. Some examples of performance requirements are as follows: the response time of a service has to be less than 3 seconds, the throughput of a service has to be greater than 10 requests/second, the utilisation of a hardware device shall not be higher than 80%, etc. Performance requirements will be used to interpret the results from the model-based performance analysis. In the *modelling* phase, an annotated[1] software architectural model is built. In the *analysis* phase, a performance model is obtained through model transformation, and such model is solved to obtain the performance results of interest.

The focus of this paper is on the *interpretation* phase where the performance results must be interpreted in order to detect, if any, performance flaws[2] and

[1] Annotations are aimed at specifying information to execute performance analysis such as the incoming workload, service demands, hardware characteristics, etc.

[2] A performance 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".

highlight the software architectural model elements responsible for that bad values. In fact, in case of unsatisfactory results a set of architectural refactoring actions can be introduced to generate new software architectural models[3] that undergo the same process shown in Figure 1.

The goal of our approach is to trace model elements vs analysis results, see shaded boxes of Figure 1. 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 foot print graph (after antipattern-based rules). This latter step is supported by performance antipatterns [5] that are suitable to deduce the logical consequences of the uncertainties, and contribute to automatically generate traces joining architectural elements and performance results.

3.1 Automatic Trace Generation

The automatic trace generation operational step (see Figure 1) takes as input: (i) performance requirements, (ii) annotated software architectural model, and (iii) performance results. It provides as output a weighted footprint graph.

Performance requirements are classified on the basis of the performance indices they address and the level of abstraction they apply. Here we consider the requirements that refer to the following performance indices [23]: *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: `RT` and `TH` of services, `U`, `QL`, and `WT` of hardware devices.

[3] We do not detail the refactoring process here, as it is out of this paper focus. However, readers interested to this part can refer to [22].

Performance 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 [23].

Annotated software architectural models may be constituted by elements belonging to different views [24]: *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) includes the specification of 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 iteractions onto hardware devices. Summarizing, `SWop`, `SWcomp`, `DYmsg`, `HWnode`, and `HWnet` represent the architectural elements we consider for traceability.

Language for Expressing Traceability. This paper adapts the language for model-to-code traceability introduced in [4] 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.

Each construct is defined as $\{m^*\}$ relationship $\{r^*\}$ where $\{m^*\}$ is the set of model elements and $\{r^*\}$ is the set of results elements. The star symbol (*) expresses multiplicity in that m* stands for multiple model elements and 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*} affectAtLeast {r*} defines that the model elements in {m*} affect all of the result elements in {r*} and possibly more. This input has a correctness constraint ensuring that every model element in {m*} individually must be affecting a subset of {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 response time (RT) and the throughput (TH) of the service S.
Input: {`SWop`*, `SWcomp`*} affectAtLeast {`RT`, `TH`}

2) AffectAtMost Construct: the input {m*} affectAtMost {r*} defines that the model elements in {m*} affect some of the result elements in {r*} but certainly not more. This input expresses the certainty that every other model element not in {m*} must not affect any result element in {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 response time (RT) and the throughput (TH) of the service S.
Input: {`SWop`*, `SWcomp`*, `HWnode`*} affectAtMost {`RT`, `TH`}

3) AffectExactly Construct: the input {m*} affectExactly {r*} defines that every model element in {m*} affects one or more result elements in {r*} and

that the results elements in {r*} are not affected in any other model element not in {m*}. This input defines no-trace between each result element in {r*} and each model element in the remaining M-{m*} (where M is the set of all input model elements), since each model element in {m*} affects only a subset of {r*}. However, this does not mean that these result elements could not be affected by other model elements in M-{m*}. 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`}

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 1.1 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 response time (RT) and the throughput (TH) of the service *S*.

```
{HWnode, HWnet} affectExactly {U, QL, WT};
{SWop, SWcomp} affectAtLeast {RT, TH};
{SWop, DYmsg} affectAtMost {RT, TH, QL};
{SWcomp, DYmsg} affectAtMost {RT, TH, QL};
```

Listing 1.1. Input to trace generation.

The goal of our SoPeTraceAnalyzer tool [7] 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).

Figure 2 reports one abstract example of this graph and it refers to the input specified in Listing 1.1. 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`), and waiting time (`WT`). 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 Figure 2 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 (ω) that represents a value indicating how much the requirement is far from the analysed index, whereas each model element node *ME* has a weight that is a function ($\sum$ F(ω))

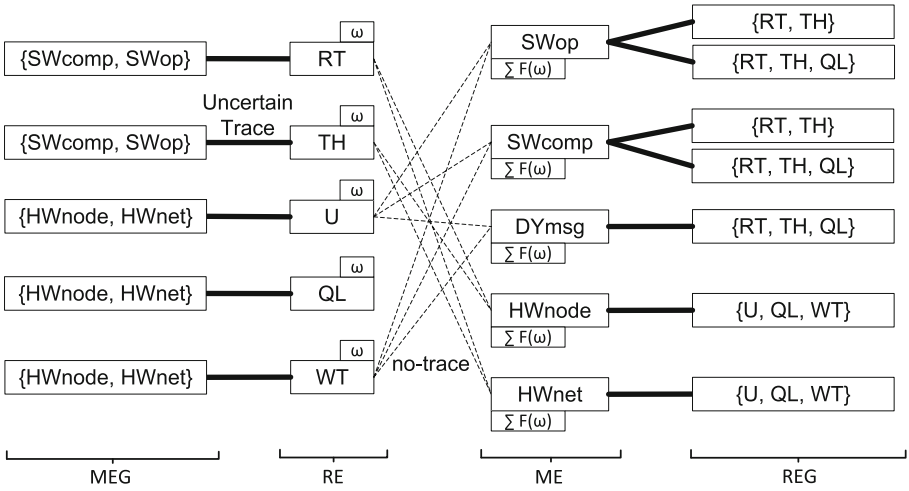


Fig. 2. Weighted Footprint Graph (from Input).

indicating how much the architectural element is critical for the violated requirements. Different heuristics (ω) and functions ($\sum$ 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 performance results and to specify legacy constraints for architectural elements. We provide preliminary heuristics in Section 4, and we intend to further investigate this aspect in the near future.

3.2 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 [5] to deduce the logical consequences between architectural elements and analysis results.

In our previous work [6] we provided a logic-based representation of performance antipatterns that supports the specification of further input to trace generation. Listing 1.2 reports the traceability rules while considering the specification of some performance antipatterns, i.e., Concurrent Processing Systems (CPS), Pipe & Filter (P&F), God Class/Component (BLOB), Extensive Processing (EP), Empty Semi Trucks (EST), One-Lane Bridge (OLB), and The Ramp (TR), respectively.

```
CPS: {HWnode} affectExactly {QL, U};
BLOB: {SWop, DYmsg} affectAtLeast {U};
P&F: {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};
```

Listing 1.2. Antipattern-based rules to reduce model-to-results uncertainty.

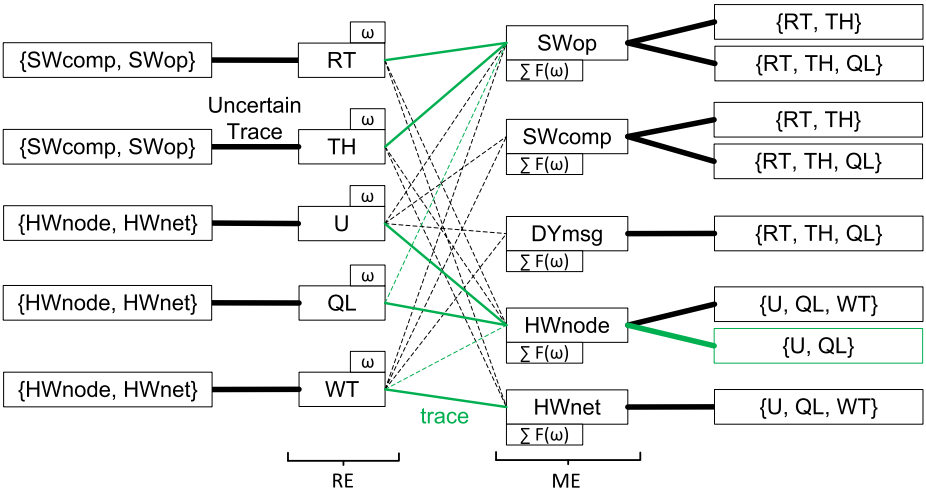


Fig. 3. Weighted Footprint Graph (after Antipattern-based Rules).

For example, detecting a CPS antipattern indicates that `HWnode` affects exactly `QL` and `U`. This rule comes from the logic-based formula of the CPS antipattern that has been defined in [6], and an excerpt is reported in Equation (1) where $\mathbb{P}$ represents the set of all the hardware devices. 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[4].

$$\exists P_x \in \mathbb{P} \mid F_{maxQL}(P_x) \geq Th_{maxQL} \wedge \\ F_{maxHwUtil}(P_x) \geq Th_{maxUtil} \tag{1}$$

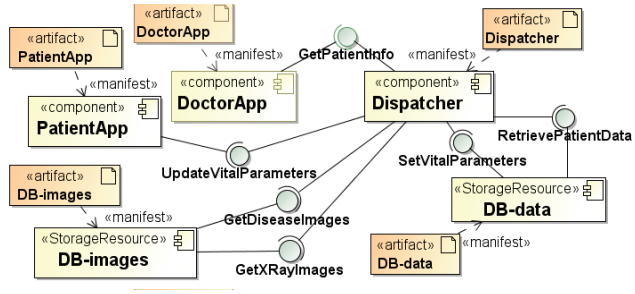
Figure 3 reports the weighted footprint graph (after introducing antipattern-based rules) and, for figure readibility, it is built considering CPS and TR rules only (see Listing 1.2). The inclusion of these two antipatterns generates five additional traces (bold lines) and two no-traces (dashed lines) between MEs and REs. Note that the specification of antipattern-based rules may also contribute to increase the overall uncertainty of the system since no logical consequences can be deduced while considering the addition of further constructs.

4 Illustrative Example

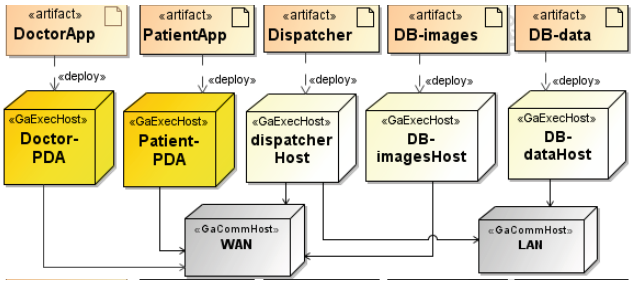
The proposed approach is illustrated on a case study in the e-health domain. Figure 4 depicts an excerpt of the E-Health System (EHS) software architectural

[4] 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 [6].

model. The system supports the doctors' everyday activities, such as the retrieval of information of their patients. On the basis of such data doctors may send an alarm in case of warning conditions. Patients are allowed to retrieve information about the doctor expertise and update some vital parameters (e.g., heart rate) to monitor their health status.



(a) Component Diagram.



(b) Deployment Diagram.

Fig. 4. EHS- Software Architectural Model.

The Component Diagram shown in Figure 4(a) describes the software components: *PatientApp* and *DoctorApp* components are connected to the *Dispatcher* component that forwards users' requests to the *DB-data* component and/or retrieves images from the *DB-images* component. The Deployment Diagram depicted in Figure 4(b) shows that both the doctor's and the patient's applications have been deployed on a Personal Digital Assistant (PDA), i.e., a mobile device. Hardware devices communicate through different networks, i.e., wide and local area networks.

The system workload has been defined as follows: (i) a closed workload is defined for the *getPatientInfo* service, with a population of 50 doctors and an average thinking time of 5 minutes; (ii) a closed workload is defined for the *updateVitalParameters* service, with a population of 2500 patients and an average thinking time of 1 hour.

The performance requirements that we consider, under the stated workload of 2550 users (i.e., 50 doctors and 2500 patients), are:

RT: The average response time of the UpdateVitalParameters service has to be less than 60 sec;

TH: The throughput of the UpdateVitalParameters service has to be greater than 4 requests/sec;

U: The utilization of the hardware devices has to be lower than 70%.

The performance analysis has been conducted by transforming the software architectural model into a Queueing Network (QN) performance model [25] and

by solving the latter with two well-assessed techniques [23], i.e., mean value analysis (MVA) and simulation. Both solution techniques are supported by Java Modeling Tools (JMT) [26]. Table 1 shows the resulting performance results for the EHS software architectural model. In particular, the average response time (RT) and throughput (TH) for the *UpdateVitalParameters* service, and the utilization (U), queue length (QL), and waiting time (WT) for the hardware devices. Shaded entries of Table 1 highlight the violated performance requirements. For example, the RT of *UpdateVitalParameters* service is predicted to be 83.51 seconds, whereas it is required to be no more than 60 seconds.

Table 1. EHS- performance analysis results.

	UpdateVital Parameters	*dispatcher Host*	*DB-data Host*	*DB-imgs Host*
$RT[sec]$	83.51	-	-	-
$TH[reqs/sec]$	3	-	-	-
$U[\%]$	-	0.18	0.93	0.32
$QL[users]$	-	0.22	12.92	0.46
$WT[sec]$	-	0.69	43.72	4.99

4.1 EHS: Automatic Trace Generation

Listing 1.3 reports one example of the set of statements that can be specified by an engineer to express the relationships between software elements and performance results in EHS, and it is provided as input to our SoPeTraceAnalyzer tool [7]. Note that such statements represent one example of engineer understanding of the system, and other feasible specifications of traceability links can be provided as well. This unavoidable gap, that recurs in any specification task, requires a wider investigation to consolidate the definition of traceability links and is left for future work.

```
{HWdbDataHost, HWwan} affectExactly
    {UdbDataHost, QLdbDataHost, WTdbDataHost};
{SWuVP, SWdbData} affectAtLeast {RTuVP, THuVP};
{SWuVP, DYsetVP} affectAtMost
    {RTuVP, THuVP, QLdbDataHost};
{SWdbData, DYsetVP} affectAtMost
    {RTuVP, THuVP, QLdbDataHost};
```

Listing 1.3. EHS- Input to trace generation.

The weighted footprint graph (from input) for EHS has been automatically obtained with the SoPeTraceAnalyzer tool [7], according to the provided specification. In particular, RE nodes are all the performance results elements of interest: response time and throughput of the *UpdateVitalParameters* service (`RTuVP`, `THuVP`), utilization, queue length, and waiting time of the *DB-dataHost* device (`UdbDataHost`, `QLdbDataHost`, `WTdbDataHost`). ME nodes are all the architectural model elements involved in the *UpdateVitalParameters* service: software operations and components (`SWuVP`, `SWdbData`), dynamic interactions (`DYsetVP`), hardware nodes (`HWdbDataHost`), and communication networks (`HWwan`).

4.2 EHS: Automatic Uncertainty Reduction

Performance antipatterns have been detected by means of our rule-based engine [6], and we found the following two instances: (i) Concurrent Processing System (CPS) antipattern, i.e., *DB-dataHost* hardware device is over-utilized; (ii) The Ramp (TR) antipattern, i.e., the response time of the *UpdateVitalParameters* service is quite unstable along simulation time.

Listing 1.4 reports the set of statements specifying the relationships between software elements and performance results in EHS, as captured by performance antipatterns. Such statements contribute to the input provided to our SoPeTraceAnalyzer tool [7].

```
CPS: {HWdbDataHost} affectExactly
     {QLdbDataHost, UdbDataHost};
TR: {SWuVP} affectExactly {RTuVP, THuVP};
```

Listing 1.4. EHS- Antipattern-based rules.

Figure 5 reports the weighted footprint graph (after antipattern-based rules) for EHS that has been automatically obtained with the SoPeTraceAnalyzer tool [7], after elaborating the rules provided by performance antipatterns.

The weight of RE nodes contribute to indicate the severity of the corresponding requirement's violation by quantifying the percentage gap between the requirement and the analysed index. Figure 5 shows that: `RTuVP` is 28% larger than the defined requirement of 60 seconds, `THuVP` is 25% lower than the defined requirement of 4 requests/sec, and `UdbDataHost` is 25% larger than the defined requirement of 70%. In fact, the *UpdateVitalParameters* service has an average response time of 83.51 sec, an average throughput of 3 requests/sec, and the utilization of the *DB-dataHost* device is 93% (see Table 1).

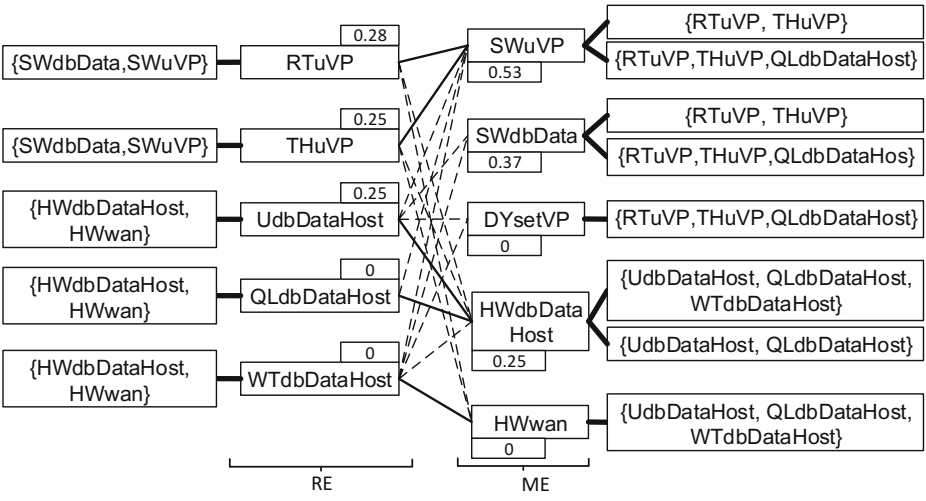


Fig. 5. EHS- Weighted Footprint Graph (after Antipattern-based Rules).

For sake of readability Figure 5 does not report the weights on the traceability links connecting RE and ME elements, however their values contribute to the weights of architectural model elements as follows. The `SWuVP` node is weighted with the value 0.53 (calculated as $0.28 * 1 + 0.25 * 1$) where the weight of 1 is assigned to the two links connecting the `SWuVP` node with `RTuVP` and `THuVP` nodes, respectively. Similarly, the `HWdbDataHost` node is weighted with the value 0.25 (calculated as $0.25 * 1$) where the weight of 1 is assigned to the link connecting `HWdbDataHost` with the `UdbDataHost` node. The `SWdbData` node is weighted with the value 0.37 (calculated as $0.28 * 0.44 + 0.25 * 1$) where the weights of 0.44 and 1 are assigned by using the guilt-based approach we defined in our previous work [11]. 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 participating. In EHS the RT of the *SWdbData* component is equal to 46.14 sec (i.e., 44% of `RTuVP` is provided by such component), whereas the TH is equal to 3 requests/sec, hence it is fully involved in the TH requirement. We recall that the *UpdateVitalParameters* service has an average response time of 83.51 sec and throughput of 3 requests/sec (see Table 1).

Table 2. EHS- performance analysis results while refactoring software model elements.

(a) *SWuVP* refactoring.

	UpdateVital Parameters	*dispatcher Host*	*DB-data Host*	*DB-imgs Host*
$RT[sec]$	35.75	-	-	-
$TH[reqs/sec]$	6	-	-	-
$U[\%]$	-	0.35	0.51	0.17
$QL[users]$	-	0.56	1.05	0.2
$WT[sec]$	-	0.87	1.63	4.37

(b) *SWdbData* refactoring.

	UpdateVital Parameters	*dispatcher Host*	*DB-data Host*	*DB-imgs Host*
$RT[sec]$	41.23	-	-	-
$TH[reqs/sec]$	5	-	-	-
$U[\%]$	-	0.18	0.77	0.04
$QL[users]$	-	0.22	3.52	0.03
$WT[sec]$	-	0.7	11.3	3.72

(c) *HWdbDataHost* refactoring.

	UpdateVital Parameters	*dispatcher Host*	*DB-data Host*	*DB-imgs Host*
$RT[sec]$	82.9	-	-	-
$TH[reqs/sec]$	3	-	-	-
$U[\%]$	-	0.22	0.36	0.32
$QL[users]$	-	0.22	0.58	0.46
$WT[sec]$	-	0.71	1.82	48.3

RE and ME nodes related to undefined and/or inviolate requirements are weighted with a value equal to zero (e.g., `QLdbDataHost`, `HWwan`).

Several strategies can be devised to use the weighted footprint graph: (i) *RE-based traceability*, i.e., looking at RE nodes only it is possible to identify the ME that most likely contribute to each requirement violation by selecting the traceability link with the highest weight; (ii) *ME-based traceability*, i.e., looking at ME nodes only it is possible to identify the most critical causes of performance flaws by providing a link coverage for all the violated requirements.

We performed a preliminary validation of these traceability strategies while separately refactoring all the model elements with a consistent weight.

Table 2 shows the performance analysis results we obtained: Table 2(a) demonstrates that the `SWuVP` refactoring is actually beneficial to solve performance flaws, since all the stated requirements have been fulfilled; Table 2(b) shows that refactoring the `SWdbData` element is beneficial for the requirements to which it is strictly connected (see Figure 5) but there is still one requirement that is not satisfied and the `SWuVP` refactoring outperforms this refactoring; Table 2(c) finally reports that refactoring the `HWdbDataHost` element is beneficial only for the requirement to which it is strictly connected (see Figure 5).

We are aware that this is far from being a rigorous proof of the weighted footprint graph output soundness, but first validation results seem promising to track a direction for this goal.

5 Discussion

Besides inheriting all limitations of the underlying software performance engineering and model-driven traceability techniques [27,28], 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.
- *Granularity:* it is difficult to establish at what level of granularity traces between model and results should be generated. Performance indices can be estimated at different levels of granularity, e.g. the response time index 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 results at all levels of abstraction.

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.

Note that our approach makes use of performance antipatterns to deduce the logical consequences between the architectural elements and analysis results,

however it does not a priori guarantee uncertainty reduction. As future work, we plan to integrate other approaches to derive model-to-results traceability links, e.g. bottleneck analysis [29] and model optimization methods [30] can be used to improve the uncertainty reduction.

6 Conclusion

This paper presents a new approach to automate the traceability between architectural model elements and performance analysis results, thus to support software architects in the identification of the causes that most likely contribute to the violation of performance requirements. To this end, we developed a tool (SoPeTraceAnalyzer) that is able to interpret a language capable of interpreting uncertainties while capturing model-to-results traceability links. The approach is illustrated by means of a case study in the e-health domain.

The benefit of the tool is that it allows to automatically visualize the dependencies between modelling elements in architectural models (e.g., software components) and performance analysis results (e.g., 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 performance antipatterns which express further such dependencies. The detection of performance antipatterns is used to make the domain expert dependencies more precise, e.g., by ruling out certain influences.

As future work we intend to apply our approach to other case studies, possibly coming from industrial experiences and different domains. This wider experimentation will allow us to deeply investigate the usefulness of performance antipatterns to reduce traceability uncertainty, thus studying the effectiveness and the scalability of our approach.

References

1. Smith, C.U., Woodside, M.: Performance validation at early stages of software development. In: System Performance Evaluation: Methodologies and Applications. CRC Press (1999)
2. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley (2002)
3. Cortellessa, V., Di Marco, A., Inverardi, P.: Model-Based Software Performance Analysis. Springer (2011)
4. Ghabi, A., Egyed, A.: Exploiting traceability uncertainty between architectural models and code. In: Joint Working IEEE/IFIP WICSA/ECSA. pp. 171–180 (2012)
5. Smith, C.U., Williams, L.G.: More new software antipatterns: even more ways to shoot yourself in the foot. In: International CMG Conference, pp. 717–725 (2003)
6. Cortellessa, V., Di Marco, A., Trubiani, C.: An approach for modeling and detecting software performance antipatterns based on first-order logics. Software and System Modeling **13**, 391–432 (2014)

7. Trubiani, C., Ghabi, A., Egyed, A.: (SoPeTraceAnalyzer). http://www.sea.uni-linz.ac.at/tools/TraZer/SoPeTraceAnalyzer.zip
8. Woodside, C.M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Workshop on the Future of Software Engineering FOSE, pp. 171–187 (2007)
9. Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., Trubiani, C.: Digging into UML models to remove performance antipatterns. In: ICSE Workshop QUOVADIS, pp. 9–16 (2010)
10. Cortellessa, V., De Sanctis, M., Di Marco, A., Trubiani, C.: Enabling performance antipatterns to arise from an adl-based software architecture. In: Joint Working IEEE/IFIP Conference WICSA/ECSA, pp. 310–314 (2012)
11. Trubiani, C., Koziolek, A., Cortellessa, V., Reussner, R.: Guilt-based handling of software performance antipatterns in palladio architectural models. Journal of Systems and Software **95**, 141–165 (2014)
12. Antoniol, G.: Design-code traceability recovery: selecting the basic linkage properties. Science of Computer Programming **40**, 213–234 (2001)
13. Egyed, A., Grunbacher, P.: Automating requirements traceability: beyond the record & replay paradigm. In: International Conference on Automated Software Engineering (ASE), pp. 163–171. IEEE (2002)
14. Cleland-Huang, J., Settimi, R., Romanova, E., Berenbach, B., Clark, S.: Best practices for automated traceability. Computer **40**, 27–35 (2007)
15. Ghabi, A., Egyed, A.: Exploiting traceability uncertainty among artifacts and code. accepted for Journal of Systems and Software (JSS) (to appear, 2014)
16. Fritzsche, M., Johannes, J., Zschaler, S., Zherebtsov, A., Terekhov, E.: Application of tracing techniques in model-driven performance engineering. In: European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA) (2008)
17. Petriu, D.B., Amyot, D., Woodside, C.M., Jiang, B.: Traceability and evaluation in scenario analysis by use case maps. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 134–151. Springer, Heidelberg (2005)
18. Alhaj, M., Petriu, D.C.: Traceability links in model transformations between software and performance models. In: Khendek, F., Toeroe, M., Gherbi, A., Reed, R. (eds.) SDL 2013. LNCS, vol. 7916, pp. 203–221. Springer, Heidelberg (2013)
19. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.: Relax: Incorporating uncertainty into the specification of self-adaptive systems. In: IEEE International Conference on Requirements Engineering, pp. 79–88 (2009)
20. Esfahani, N., Malek, S., Razavi, K.: Guidearch: guiding the exploration of architectural solution space under uncertainty. In: International Conference on Software Engineering (ICSE), pp. 43–52 (2013)
21. Letier, E., Stefan, D., Barr, E.T.: Uncertainty, risk, and information value in software requirements and architecture. In: International Conference on Software Engineering (ICSE), pp. 883–894 (2014)
22. Arcelli, D., Cortellessa, V., Trubiani, C.: Antipattern-based model refactoring for software performance improvement. In: International Conference on Quality of Software Architectures (QoSA), pp. 33–42 (2012)
23. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. SIGMETRICS Performance Evaluation Review **19**, 5–11 (1991)

24. Clements, P.C., Garlan, D., Little, R., Nord, R.L., Stafford, J.A.: Documenting software architectures: views and beyond. In: International Conference on Software Engineering (ICSE), pp. 740–741 (2003)
25. Cortellessa, V., Mirandola, R.: Prima-uml: a performance validation incremental methodology on early uml diagrams. Sci. Comput. Program. **44**, 101–129 (2002)
26. Casale, G., Serazzi, G.: Quantitative system evaluation with java modeling tools. In: International Conference on Performance Engineering (ICPE), pp. 449–454 (2011)
27. Smith, C.U.: Introduction to software performance engineering: origins and outstanding problems. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 395–428. Springer, Heidelberg (2007)
28. Gotel, O., Cleland-Huang, J., Hayes, J.H., Zisman, A., Egyed, A., Grünbacher, P., Antoniol, G.: The quest for ubiquity: a roadmap for software and systems traceability research. In: IEEE International Requirements Engineering Conference (RE), pp. 71–80 (2012)
29. Franks, G., Petriu, D.C., Woodside, C.M., Xu, J., Tregunno, P.: Layered bottlenecks and their mitigation. In: International Conference on the Quantitative Evaluation of Systems (QEST), pp. 103–114 (2006)
30. Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: A systematic literature review. IEEE Trans. Software Eng. **39**, 658–683 (2013)