Exploiting Architecture/Runtime Model-driven Traceability for Antipattern-based Performance Improvement

Davide Arcelli, Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo and Michele Tucci

March 8, 2019

Exploiting Architecture/Runtime Model-driven Traceability for Performance Improvement

Davide Arcelli, Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo, Michele Tucci
Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy
{davide.arcelli, vittorio.cortellessa, daniele.dipompeo, romina.eramo, michele.tucci}@univaq.it

***Abstract*—Model-Driven Engineering techniques may achieve a major support to the software development when they allow to manage relationships between a running system and its architectural model. These relationships can be exploited for different goals, such as the software evolution due to new functional requirements. In this paper, we define and use relationships that work as support to the performance improvement of a running system. In particular, we combine: (i) a bidirectional model transformation framework tailored to define relationships between performance monitoring data and an architectural model, with (ii) a technique for detecting performance antipatterns and for suggesting architectural changes, aimed at removing performance problems identified on the basis of runtime information. The result is an integrated approach that exploits traceability relationships between the monitoring data and the architectural model to derive recommended refactoring solutions for the system performance improvement. The approach has been applied to an e-commerce application based on microservices that has been designed by means of UML software models profiled with MARTE.**

***Index Terms*—Software Performance, Architecture traceability, Model-driven engineering**

I. Introduction

Over the last decades, the fast growing complexity of software systems has forced practitioners to use and investigate different development techniques to tackle advances in productivity and quality. To this extent, software engineering needs to relay on automated approaches to keep low the development costs while tackling the rapid changes of software capabilities that expose different non-functional properties.

In order to manage software complexity, ever more companies are considering Model-Driven Engineering (MDE) [1] approaches, with the perceived benefit of enabling developers to work at a higher level of abstraction and to rely on automation throughout the development process. Nevertheless, MDE solutions need to be further developed to scale up for real-life industrial projects [2]. To this intent, one of the major challenges is to work on achieving a more efficient integration between the design and runtime aspects of systems. For instance, through observation and instrumentation, logs and metrics can be collected and related to the original software design in order to comprehend, extrapolate and analyze the inner behavior of running software system [3].

In support of this, a recent European project[1] has been founded and supported by both industry and academic partners. As part of its continuous system engineering approach [4], the project notably aims at providing a runtime-design time feedback loop that could be deployed and used in different industrial domains. Such a feedback from runtime to architectural design level can certainly be exploited to let the developers have some sort of control and manipulation possibilities over elements they would not be able to access otherwise.

In this context, non-functional properties (e.g. performance, power consumption or memory footprint) are becoming ever more relevant for the success of a software application, and the early identification of problems induces lower cost solutions [5]. On one side, in model-based software performance engineering, a number of approaches have been proposed for detecting and removing performance problems in software models. Some techniques are based on the concept of performance antipattern, which characterizes bad design practices that may jeopardize software performance, along with possible refactoring actions aimed to remove them [6]. On the other side, methods and tools have been proposed for monitoring system execution and measuring performance of running systems. However, many of them do not envisage a solid integration with architectural design models [2]. Instead, one of the main benefits in adopting model-based performance evaluation is the ability to conduct analysis (e.g., what-if analysis) that would be expensive on a real system, such as to analyze the system behavior when subjected to different workloads, or to analyze the performance sensitivity to system parameter variations within some ranges. Basing on a solid connection between runtime information and architectural design, developers can suggest architectural changes needed to meet performance requirements before the system actually experiments certain scenarios (e.g., some specific workloads).

This paper proposes a model-driven approach to support designers in their performance analysis and model refactoring processes by exploiting design/runtime interactions. In particular, the system behavior at runtime is related to the architectural design in order to investigate potential performance issues in design and to suggest possible model refactoring actions. The approach has been realized within Eclipse EMF[2]

This research was supported by the ECSEL-JU through the MegaM@Rt2 project (grant agreement No 737494).

[1]MegaM@Rt2 project: https://megamart2-ecsel.eu/
[2]Eclipse Modeling Framework: https://www.eclipse.org/modeling/emf/

and it integrates a model-driven framework for the definition of correspondences between design and runtime with another one for performance analysis and model refactoring, respectively JTL [7] and PADRE [8].

The main novel contributions of this paper are: (i) the definition of a traceability model between logs extracted from a running system and an architectural model, (ii) feeding an architectural model with runtime monitored performance indices, (iii) engineering an end-to-end solution for performance improvement based on the interoperability between JTL and PADRE.

The approach has been applied on an e-commerce application based on microservices that has been designed by means of UML software models (profiled with MARTE) [9, 10].

The rest of the paper is organized as follows: Section II introduces some background information on the model-driven techniques that have been used in the work. Section III presents the overall approach. Section IV describes the case study that is used in Section V to show the approach in practice. Then, related work is presented in Section VI, whereas Section VII concludes the paper.

II. Background

Model Driven Engineering (MDE) [1] leverages intellectual property and business logic from source code into high-level specifications enabling more accurate analyses. In general, an application domain is consistently analyzed and engineered by means of a *metamodel*, i.e., a coherent set of interrelated concepts. A model is said to *conform* to a metamodel, meaning that the former is expressed by the concepts encoded in the latter. Constraints are defined at the meta-level, and the consistency relationships between models are guaranteed by means of (bidirectional) model transformations specified on source and target metamodels. With the introduction of model-driven techniques in the software lifecycle, also the analysis of non-functional properties has became effective by means of dedicated tools for the automated assessment of quality attributes [11].

This paper proposes a model-driven approach to support designers in a performance analysis process that involves model-driven traceability links to relate software architecture and runtime information (as introduced in Sect. I). To this end, the approach exploits the following frameworks.

A. JTL

JTL (Janus Transformation Language) [7] is an Eclipse EMF-based tool realized to maintain consistency between software artifacts. Its constraint-based and relational model transformation engine is specifically tailored to support bidirectionality and change propagation. The JTL transformation mechanism provides a relational semantics relying on Answer Set Programming (ASP) [12]. The bidirectional engine provides the possibility to apply the transformation rules in both ways, from right to left domains and viceversa. Furthermore, given a change to one model (e.g., the target model), JTL uses the DLV constraint solver [13] to find a consistent choice for the other model (eg., the source model).

The traceability mechanism stores relevant details about the linkage between right and left model elements at execution-time (including the applied transformation rules) [14]. Traceability links are extrapolated during the transformation execution and made explicit by the framework. In fact, traceability models are maintained as models conforms to the dedicated traceability metamodel, as defined in its Ecore format within EMF. Traceability models can be stored, viewed and manipulated (if needed) by the designer. Moreover, traceability models can be re-used during the transformation execution: they can be given as input of the transformation in order to (re-)establish consistency, manage ambiguities and guarantee the correctness of the transformation.

B. PADRE

PADRE (Performance Antipatterns Detection and model REfactoring) [8] is an Eclipse-based framework that provides an integrated environment for the detection of performance antipatterns (i.e., bad practices that might cause performance degradation) and the application of refactoring actions to UML models properly profiled with MARTE for sake of performance indices and parameters annotation. In this context, refactoring goal is to improve software performance while removing performance antipatterns occurrences. PADRE is equipped with a set of performance antipatterns detection rules based on those presented in [15], which are verified in order to identify bad design practices that might lead to performance degradation; subsequently, PADRE is able to apply predefined refactoring actions aimed at removing the identified antipatterns occurrences.

PADRE exploits the Epsilon platform [16] to provide several kinds of detection and refactoring sessions with different levels of automation, which are based on Epsilon Validation Language (EVL), Epsilon Pattern Language (EPL), and Epsilon Wizard Language (EWL), respectively. The first one is used in this paper, providing so-called *user-driven sessions*, where a list of performance antipatterns occurrences is presented to the user, which can apply one (or more) predefined refactoring action(s) among the available ones. After a refactoring action has been applied, PADRE automatically executes performance analysis (by exploiting a derived performance model) and provides feedback to the user in order to decide whether the obtained refactored model meets performance requirements.

III. Overall approach

The idea underlying our approach exploits the correspondences between the architectural design and the runtime aspects of a software system, with the aim of improving its performance. In particular, the system behavior at runtime is matched with the architectural design in order to connect performance critical situations at runtime with their corresponding potential causes in design. We specify such design-runtime correspondences by means of traceability models to make the relationships between system design and runtime information consistent and exploitable. Consequently, the analysis of such traceability models can help to discover system properties deviations and to identify the affected components.

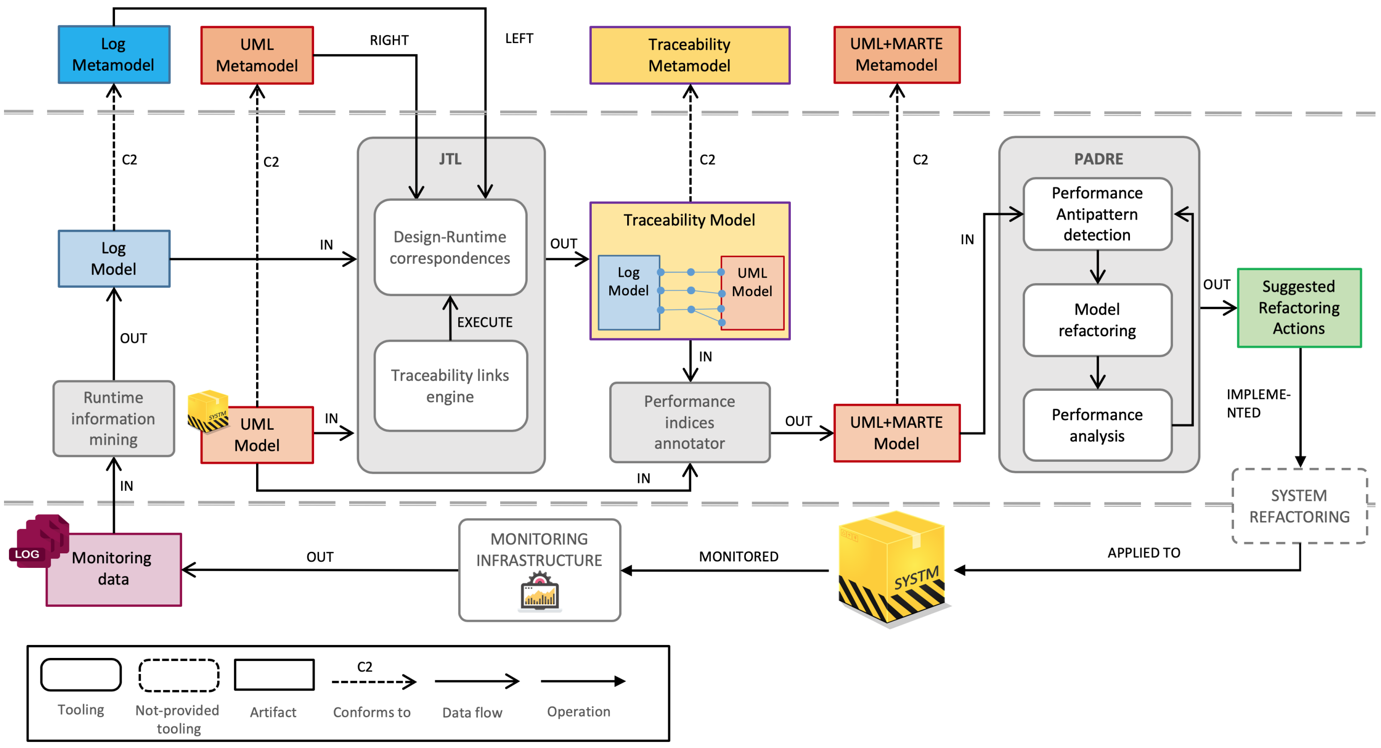


Fig. 1: Overall approach

The overall architecture of our approach is depicted in Fig. 1, where the top side represents the flow of activities on design artefacts, whereas the bottom side contains runtime activities. The large gray boxes represent the *JTL* and the *PADRE* frameworks that have been introduced in the previous section. The system under analysis is represented by the yellow box in the bottom part of the figure. The flow starts with an architectural design (represented by an *UML Model*) and *Monitoring data* (represented as *Log* files), and it follows a sequence of steps that are described in the following.

A. Runtime information mining

As depicted in Fig. 1, runtime data (i.e., logs/traces) are obtained through a monitoring infrastructure over a running system. The specific infrastructure adopted for our case study is detailed in Section IV.

The collected runtime information is then integrated in the EMF-based environment and translated in EMF artifacts. In this step, Logs are automatically transformed in *Log Model*s that conform to a specific *Log Metamodel* (properly defined in Ecore format) representing monitoring data. In Section V-A, the specified metamodel is shown and used to automatically generate log models.

B. Design-Runtime traceability with JTL

Correspondences between the system behavior at runtime and the architectural design can be defined in several ways. They are often based on the concept of traceability relationships, which may help designers to understand associations and dependencies that exist among heterogeneous models [17, 18]. In MDE, a *traceability link* is a relationship between one or more source model elements and one or more target model elements, whereas a *trace model* is a structured set of traceability links, e.g., between source and target models.

In this work, we propose to generate traceability links between UML and Log Models by means of JTL. In particular, the tool allows to specify *Design-Runtime correspondences* in a declarative way at metamodel level, as bidirectional model transformations (i.e., between design and runtime metamodels). The JTL *Traceability engine* is able to execute such bidirectional model transformations and automatically generate the correspondent traceability links between elements of the UML design model and the log model ones. Traceability links are collected in an explicit way as in Traceability Models conforms to a dedicated metamodel, namely the reference JTL *Traceability Metamodel* described in Section V-B).

Although in this paper JTL is used only in one direction (i.e., from running logs to UML model), its bidirectional capability can be exploited for sake of system refactoring automation, namely for reporting suggested refactoring actions from the modeling level to the running level. Obviously, being UML and Traceability models at different levels of abstraction, the *Design-Runtime correspondences* are not necessarily one-to-one. It could be often the case that a single element in an UML model corresponds to multiple elements in the running system. However, the results of antipattern detection joint with composability/decomposability rules of performance indices lead enough information to treat this aspect. One possible scenario is that a single refactoring action on a model could correspond to multiple alternative actions on the running system.

C. *Performance indices annotator*

This step represents an intermediate phase where the knowledge gained above is used to enable an effective detection and resolution of performance problem within PADRE. Indeed, the correspondences between runtime information and software models are key to infer performance properties: performance data are used to annotate the UML Model by means of the MARTE profile, thus obtaining the *UML+MARTE Model* in Fig. 1.

In general, performance parameters (e.g., workload, resource demands) represent input values requested to solve the performance model derived from the software model, whereas indices (e.g., throughput and response time) represent the output of performance analysis. In this step, both parameters and indices are obtained and then annotated on the UML+MARTE Model that will be used within PADRE (as described in details later) for the sake of *Performance Antipattern detection*. For instance, the *Resource Demand D* of a software operation is obtained as the product of its *Service Time S* and *Number V of Visits* [19], where S has been estimated by stimulating the system with a lightweight load to avoid waiting time. Hence, such annotation step augments the UML Model by introducing performance-related information that is exploited for detecting performance antipattern occurrences on the basis of realistic metrics gathered from the running system.

In Section V-C we show how performance indices are calculated and back-annotated to the UML+MARTE Model of the considered case study.

D. *Performance analysis and refactoring with PADRE*

PADRE is used to perform a performance antipattern analysis and to suggest the most promising refactoring actions that shall remove detected antipatterns and then improve the overall system performance.

As illustrated in Fig. 1, starting from an UML+MARTE Model, *Performance Antipattern detection* is first executed, followed by a *Model refactoring* step. The following *Performance Analysis* step in the figure collapses two sub-steps, that are: (i) automated transformation of the refactored model into a performance model (i.e. a Queueing Network [11]), and (ii) execution of Mean-Value Analysis [19] on the performance model to obtain the current performance indices. PADRE is also able to back-annotate the UML+MARTE Model with the obtained performance indices, so that it can undergo a new refactoring loop. The process lasts until either the user stops it or no more performance antipatterns are detected on the model. The output of this process is represented by a set of well-formed refactoring actions that, at the design level, have demonstrated to effectively improve the system performance. The system refactoring step is represented as a dashed box between the design and the runtime areas of Fig. 1, because the propagation of these actions on the running system is still to be automated.

In the next section, a case study is presented and later used to show the approach at work.

IV. Case study

In this paper, we consider an e-commerce web application based on microservices, called *E-Shopper*[3]. According to the microservice architecture, the application is developed as a suite of small services, each running in its own process and communicating with RESTful HTTP API. The application is composed by 9 microservices, each requiring a different database to operate, and developed on top of the *Spring Cloud*[4] framework.

The E-Shopper application is designed by means of the UML language. All the views of the system are embedded in a single model file, as it is customary among UML graphical design tools. A fragment of the UML software model is depicted in Fig. 2. The UML Component Diagram in Fig. 2a shows an excerpt of the static view of the application by means of software components (each providing a microservice) and their interconnections. The UML Deployment Diagram in Fig. 2b depicts the deployment view of the system, ie., hardware nodes and how the application artifacts are allocated on them. Specifically, each microservice is deployed on a different *Docker*[5] container.

Among the existing Use Cases of the application, we consider three scenarios that are representative of the different purposes of the system. In particular, the *Web* scenario describes the requests to the home page that the site visitors perform via the browser. For instance, the UML Sequence Diagram in Fig. 2c illustrates the dynamic view in which the web service randomly finds items in the E-Shopper categories. Moreover, the *Mobile* scenario describes the research of a product by means of the mobile application. Finally, the *Warehouse* scenario describes the requests performed by the employees while checking the products availability. In this paper, we focused on the performance analysis of the Web scenario.

To the aim of monitoring the running application, the following infrastructure has been set up. The application has been instrumented by means of the *Spring Cloud Sleuth*[6] distributed tracing solution. Thus, the running application is monitored during a load test performed with *JMeter*[7]. The workload was designed to stress the application by simulating a significant amount of requests along the three mentioned scenarios.

The log traces produced during the execution are gathered by the *Zipkin*[8] distributed tracing system. On turn, Zipkin is configured to forward the monitoring data to the distributed database and search engine *Elasticsearch*[9]. A sample of raw logs collected within Elasticsearch is shown in Fig. 3. For instance, the first item with time *18/11/20 09:52:48.107* includes all the information related to the span with id

[3]The application is available at https://git.io/fh9Z8

[4]Spring Cloud: https://spring.io/projects/spring-cloud

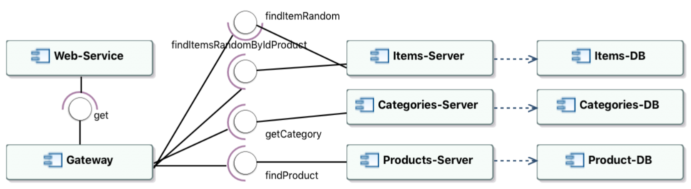
[5]Docker: https://www.docker.com

[6]Spring Cloud Sleuth: https://spring.io/projects/spring-cloud-sleuth

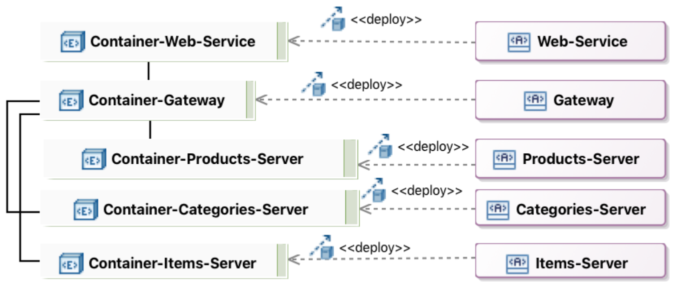
[7]Apache JMeter: https://jmeter.apache.org/

[8]Zipkin: https://zipkin.io/

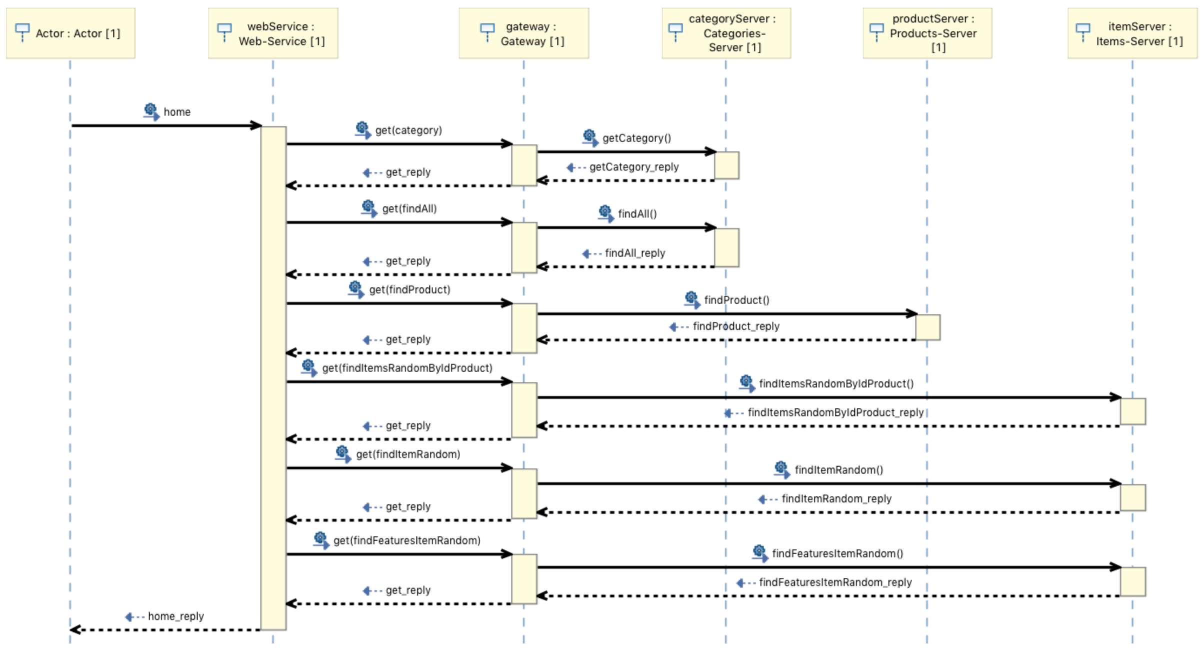
[9]Elasticsearch: https://www.elastic.co/products/elasticsearch



(a) Static view (UML Component diagram)



(b) Deployment view (UML Deployment diagram)



(c) Dynamic view (UML Sequence diagram)

Fig. 2: An excerpt of the *E-Shopper* UML Software Model

16bb4e7b689f807a that have been gathered during the monitoring (ie., trace id, duration, timestamp, service and related end point, type of span, IP address, port number, etc.).

V. The Approach in Practice

In this section we show a stepwise application of the proposed approach to an E-Shopper case study.

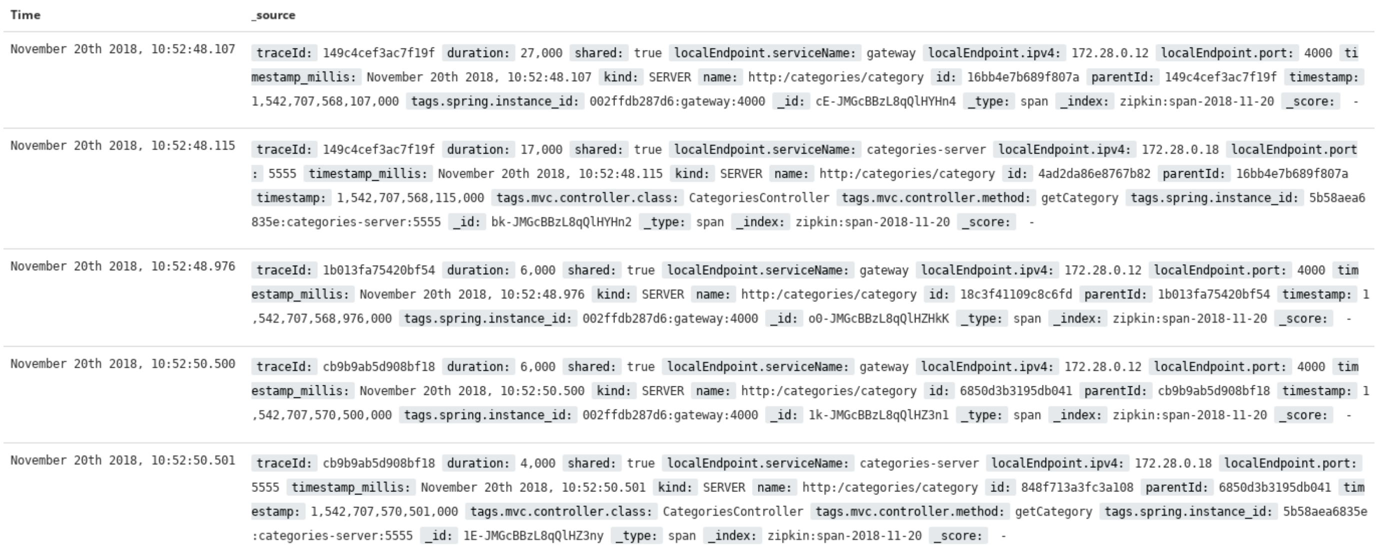
A. *Runtime information mining*

The raw logs obtained from the infrastructure described in the previous section have to be specified by means of a model-based representation. To this aim, we have defined a dedicated metamodel, as depicted in Fig. 4. It basically defines the notion of *Log*, which is the root element of a log model. A Log stores all the *Trace* information for messages being sent to *EndPoint*s representing the microservices (*Service*s) of an application. A Trace is identified by a unique *ID* and includes a set of *Spans* that represent execution events. A Span is specified by the following attributes: *timestamp* that describes when the event occurs, *duration* that describes the time to complete the call, and *kind* that is *SERVER*, *CLIENT* or undefined. A Span refers to an EndPoint that is an URL used to access the project and refers to one or more Services.

Figure 5 depicts a sample of a Log Model that represents the original logs shown in Fig. 3. For instance, the topmost Span (id *16bb4e7b689f807a*) represents the first span in Fig. 3 with a *27ms* duration, by *SERVER* kind, and with the *18/11/20 09:52:48.107* timestamp for the call to the *http://categories/category* EndPoint belonging to the *gateway* Service. Note that the information that is negligible for our purposes has not been included. The model is automatically generated from the original raw log by means of a Java transformation able to serialize the textual representation of the logs into xmi-encoded models conforms to the Log Metamodel.

B. *Design-Runtime traceability with JTL*

Starting from the Log and UML models, a Traceability Model is automatically generated by means of JTL [14]. In order to represent the traceability information between design and runtime artefacts, the JTL traceability reference metamodel is considered. As depicted in Fig. 6, it basically defines



```
Time                              _source

November 20th 2018, 10:52:48.107  traceId: 149c4cef3ac7f19f duration: 27,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 ti
                                  mestamp_millis: November 20th 2018, 10:52:48.107 kind: SERVER name: http:/categories/category id: 16bb4e7b689f807a parentId: 149c4cef3ac7f19f timestamp:
                                  1,542,707,568,107,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: cE-JMGcBBzL8qQlHYHn4 _type: span _index: zipkin:span-2018-11-20 _score: -

November 20th 2018, 10:52:48.115  traceId: 149c4cef3ac7f19f duration: 17,000 shared: true localEndpoint.serviceName: categories-server localEndpoint.ipv4: 172.28.0.18 localEndpoint.port
                                  : 5555 timestamp_millis: November 20th 2018, 10:52:48.115 kind: SERVER name: http:/categories/category id: 4ad2da86e8767b82 parentId: 16bb4e7b689f807a
                                  timestamp: 1,542,707,568,115,000 tags.mvc.controller.class: CategoriesController tags.mvc.controller.method: getCategory tags.spring.instance_id: 5b58aea6
                                  835e:categories-server:5555 _id: bk-JMGcBBzL8qQlHYHn2 _type: span _index: zipkin:span-2018-11-20 _score: -

November 20th 2018, 10:52:48.976  traceId: 1b013fa75420bf54 duration: 6,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 tim
                                  estamp_millis: November 20th 2018, 10:52:48.976 kind: SERVER name: http:/categories/category id: 18c3f41109c8c6fd parentId: 1b013fa75420bf54 timestamp: 1
                                  ,542,707,568,976,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: o0-JMGcBBzL8qQlHZHkK _type: span _index: zipkin:span-2018-11-20 _score: -

November 20th 2018, 10:52:50.500  traceId: cb9b9ab5d908bf18 duration: 6,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 tim
                                  estamp_millis: November 20th 2018, 10:52:50.500 kind: SERVER name: http:/categories/category id: 6850d3b3195db041 parentId: cb9b9ab5d908bf18 timestamp: 1
                                  ,542,707,570,500,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: 1k-JMGcBBzL8qQlHZ3n1 _type: span _index: zipkin:span-2018-11-20 _score: -

November 20th 2018, 10:52:50.501  traceId: cb9b9ab5d908bf18 duration: 4,000 shared: true localEndpoint.serviceName: categories-server localEndpoint.ipv4: 172.28.0.18 localEndpoint.port:
                                  5555 timestamp_millis: November 20th 2018, 10:52:50.501 kind: SERVER name: http:/categories/category id: 848f713a3fc3a108 parentId: 6850d3b3195db041 tim
                                  estamp: 1,542,707,570,501,000 tags.mvc.controller.class: CategoriesController tags.mvc.controller.method: getCategory tags.spring.instance_id: 5b58aea6835e
                                  :categories-server:5555 _id: 1E-JMGcBBzL8qQlHZ3ny _type: span _index: zipkin:span-2018-11-20 _score: -
```

Fig. 3: A fragment of the raw log in Elasticsearch

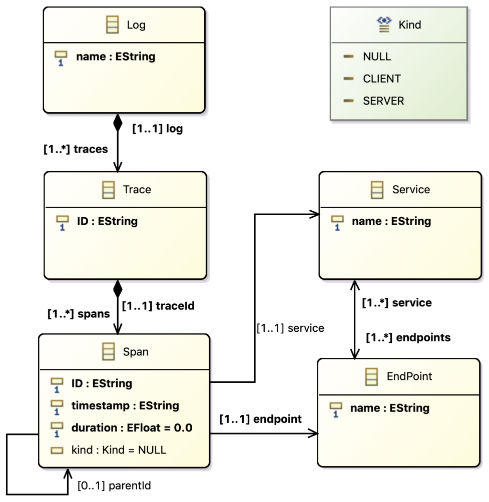


Fig. 4: Log Metamodel

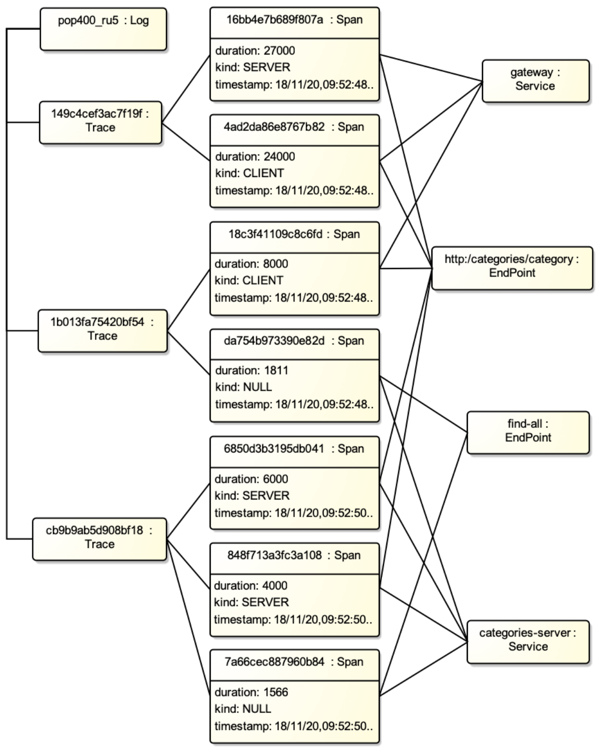


Fig. 5: A Log Model sample

the notion of *TraceModel*, which is the root element of a traceability model.

It relates a model belonging to a "left" domain to a model belonging to a "right" domain. In particular, a set of trace links between left and right elements and the rules that enforced their mapping are collected. A *TraceLink* relates one or more elements belonging to the left domain (*leftLinkEnd*) and the correspondent (one or more) elements belonging to the right domain (*rightLinkEnd*). Such links connect elements of *TraceLinkEnd* type that have a name and a type. Each TraceLinkEnd refers to an object of *EObject* type (org.eclipse.emf.ecore.EObject) that represents a specific object in the left or right domain.

The correspondences between the design and runtime concepts are defined by means of JTL and specified between the corresponding metamodels, as reported in List. 1. The specification is defined by means of *relations* between elements of the two involved *domains*. In particular, on Line 1, variables *log* and *uml* are declared to match models conforming to the Log and UML metamodels, respectively. The specified relations are described as follows:

- The top relation *Trace2UseCase* (Lines 3-13) maps a container element of Trace type in the Log domain and a container element of UseCase type in the UML domain. The *where* clause invokes the execution of the *Span2Message* relation;
- The *Span2Message* relation (Lines 14-22) maps a Span and a Message type element involved in a use case

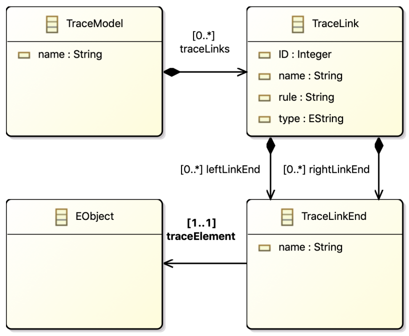


Fig. 6: JTL Traceability Metamodel

interaction. The *where* clause invokes the execution of the *EndPoint2Signature* relation;

- The *EndPoint2Signature* relation (Lines 23-31) maps an EndPoint of a Span and an Operation type element that represents the signature of a message;
- The top relation *Service2Component* (Lines 32-40) maps a Service type container element to a Component type one.

```
1  transformation Log2UML (log:Log, uml:UML) {
2    ...
3    top relation Trace2UseCase {
4      checkonly domain log t : Log::Trace {
5        spans = s : Log::Span {   }
6      };
7      checkonly domain uml uc : UML::UseCase {
8        ownedBehavior = ob : UML::Interaction {
9          message = m : UML::Message { }
10       }
11     };
12     where { Span2Message(s, m); }
13   }
14   relation Span2Message {
15     checkonly domain log s : Log::Span {
16       endpoint = ep : Log::EndPoint { }
17     };
18     checkonly domain uml m : UML::Message {
19       signature = s : UML::Operation { }
20     };
21     where { EndPoint2Signature(ep, s); }
22   }
23   relation EndPoint2Signature {
24     n : String;
25     checkonly domain log ep : Log::EndPoint {
26       name = n
27     };
28     checkonly domain uml s : UML::Operation {
29       name = n
30     };
31   }
32   top relation Service2Component {
33     n : String;
34     checkonly domain log s : Log::Service {
35       name = n
36     };
37     checkonly domain uml c : UML::Component {
38       name = n
39     };
40   }
41   ...
42 }
```

Listing 1: Log2UML correspondences specification

The described mapping assumes that models are consistent with the monitored code. In this case study, models and code are also consistent in terms of naming convention used. However, JTL allows specifying also complex relationships between elements, e.g., elements that do not trivially match by names, or model elements that do not map one-to-one to the code [7].

The application of the *Log2UML* transformation on the Log and UML models, as shown in the left and right part of Fig. 7, generates the corresponding Traceability model, as shown in the middle of that figure. In particular, the arrows connect trace links with the source and target model elements they refer to. For instance, the *Trace2UseCase_149c4cef3ac7f19f* traceability link relates the *Get HomePage* use case in the right end and the corresponding *149c4cef3ac7f19f* log trace in the left end. Hence, for each message in the use case, we are able to know when the corresponding operation has started and its response time. As a consequence, the traceability model can be used to derive complex measures such as the average response time of a specific scenario or the average service time of an operation (as mentioned in Section III-C).

C. Performance indices annotator

In this step, the Traceability Model generated above is used to extract performance properties and incorporate them into the UML Model by means of the MARTE profile. In particular, the runtime information obtained by the monitoring infrastructure is used to obtain the performance input parameters, whereas the relationships between runtime information and software model are used to identify the proper UML elements that have to be annotated with these parameters.

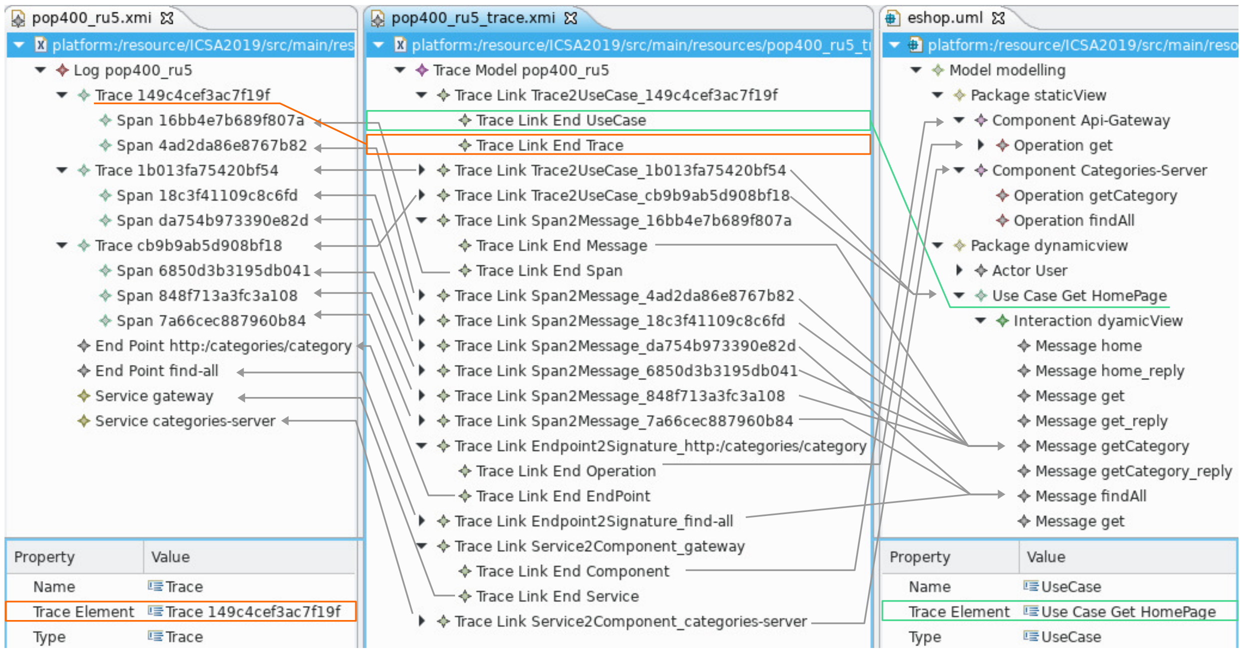
The considered performance properties have been obtained as described in Section III-C and annotated in the UML+MARTE Model as following:

- The service time S of each operation is annotated on its relative message in the scenario by means of the *servCount* attribute of the *GaAcqStep* stereotype. Also, the number of times a message occurs in the same scenario, which corresponds to the number V of visits, is annotated in the *rep* tag of the same stereotype.
- The response time of a whole scenario is annotated in the *respT* tag of the *GaScenario* stereotype; it corresponds to the response time of the parent span in the corresponding trace (see the *149c4cef3ac7f19f* Trace and the *Get Home Page* use case in the Traceability Model of Fig. 7).
- The throughput is annotated in the *throughput* attribute of the *GaScenario* stereotype.
- The workload is annotated in the *pattern* tag of *GaWorkloadEvent* stereotype, which can represent an *open* or a *closed* class of jobs.
- The utilization of a Docker container is set through the *utilization* tag of the *GaExecHost* stereotype applied to the corresponding UML Device.

An excerpt of the UML Component and Sequence diagrams annotated with MARTE is depicted in Fig. 8.

D. Performance analysis and refactoring with PADRE

In this step, the obtained UML+MARTE Model is given as input to PADRE. The Performance Antipattern detection step



```
pop400_ru5.xmi
▾ platform:/resource/ICSA2019/src/main/res
  ▾ Log pop400_ru5
    ▾ Trace 149c4cef3ac7f19f
        Span 16bb4e7b689f807a
        Span 4ad2da86e8767b82
    ▾ Trace 1b013fa75420bf54
        Span 18c3f41109c8c6fd
        Span da754b973390e82d
    ▾ Trace cb9b9ab5d908bf18
        Span 6850d3b3195db041
        Span 848f713a3fc3a108
        Span 7a66cec887960b84
      End Point http:/categories/category
      End Point find-all
      Service gateway
      Service categories-server
```

Property	Value
Name	Trace
Trace Element	Trace 149c4cef3ac7f19f
Type	Trace

```
pop400_ru5_trace.xmi
▾ platform:/resource/ICSA2019/src/main/resources/pop400_ru5_t
  ▾ Trace Model pop400_ru5
    ▾ Trace Link Trace2UseCase_149c4cef3ac7f19f
        Trace Link End UseCase
        Trace Link End Trace
    ▸ Trace Link Trace2UseCase_1b013fa75420bf54
    ▸ Trace Link Trace2UseCase_cb9b9ab5d908bf18
    ▾ Trace Link Span2Message_16bb4e7b689f807a
        Trace Link End Message
        Trace Link End Span
    ▸ Trace Link Span2Message_4ad2da86e8767b82
    ▸ Trace Link Span2Message_18c3f41109c8c6fd
    ▸ Trace Link Span2Message_da754b973390e82d
    ▸ Trace Link Span2Message_6850d3b3195db041
    ▸ Trace Link Span2Message_848f713a3fc3a108
    ▸ Trace Link Span2Message_7a66cec887960b84
    ▾ Trace Link Endpoint2Signature_http:/categories/category
        Trace Link End Operation
        Trace Link End EndPoint
    ▸ Trace Link Endpoint2Signature_find-all
    ▾ Trace Link Service2Component_gateway
        Trace Link End Component
        Trace Link End Service
    ▸ Trace Link Service2Component_categories-server
```

```
eshop.uml
▾ platform:/resource/ICSA2019/src/main/reso
  ▾ Model modelling
    ▾ Package staticView
      ▾ Component Api-Gateway
        ▸ Operation get
      ▾ Component Categories-Server
          Operation getCategory
          Operation findAll
    ▾ Package dynamicview
      ▸ Actor User
      ▾ Use Case Get HomePage
        ▾ Interaction dyamicView
            Message home
            Message home_reply
            Message get
            Message get_reply
            Message getCategory
            Message getCategory_reply
            Message findAll
            Message get
```

Property	Value
Name	UseCase
Trace Element	Use Case Get HomePage
Type	UseCase

Fig. 7: Traceability model between the *E-Shopper* Runtime and UML models

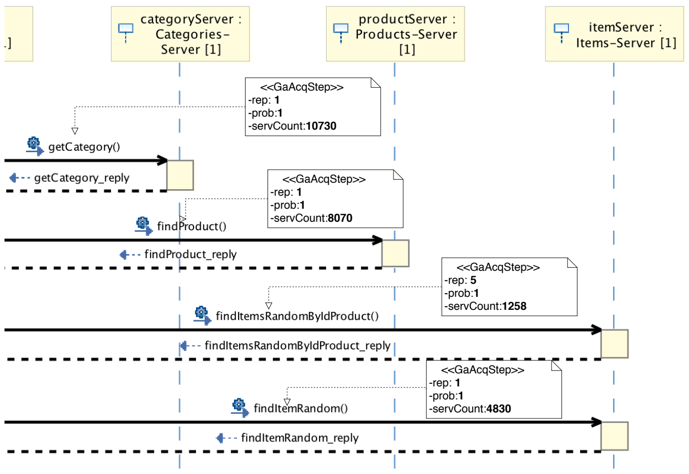
on our case study has identified the following performance antipatterns:

- two occurrences of the *Pipe and Filter (PaF)* performance antipattern due to the extensive processing of critical operations in the *Web Service* and *Items Server* microservices, respectively. In particular, a *Pipe and Filter* highlights that, among the calling methods (i.e., chain of messages), an operation's demand is larger than a threshold and, as a consequence, the platform is overutilized;
- an occurrence of the *One-Lane Bridge (OLB)* performance antipattern due to a congestion on the *Web Service* microservice. In particular, the microservice receives a number of calls greater than its pool size, thus the response time of the considered scenario goes over a predefined threshold.

PADRE proposes a set of refactoring actions for each detected performance antipattern. For instance, among the actions proposed for a PaF performance antipattern, PADRE suggests to move the most demanding operations either to a new component deployed on a new node or to a component with less demanding operations. Moreover, PADRE suggests to deploy the critical component on a new and more powerful node, or on the node with the lowest utilization.

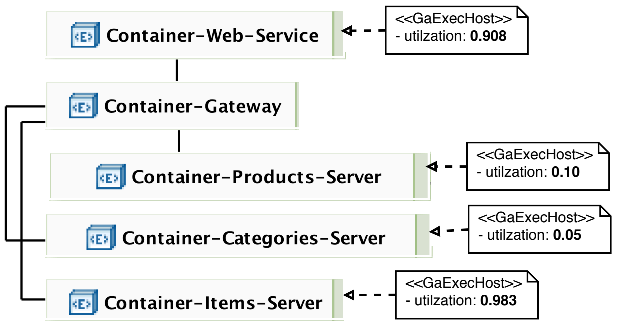
In Table I we have summarized the results obtained from our refactoring session: the first column reports the type of refactoring, whereas the obtained response time and its improvement are reported for each scenario.

First, we have applied a refactoring action on the *Items Server* component. In particular, we decided to move the most demanding operation *findProduct()* of *Items Server* to a new component (i.e., the new *Items-Server-2* microservice) deployed on a new node (i.e., the new *Items-Server-*



categoryServer : Categories-Server [1]
productServer : Products-Server [1]
itemServer : Items-Server [1]
<<GaAcqStep>> -rep: 1 -prob:1 -servCount:10730
getCategory()
getCategory_reply
<<GaAcqStep>> -rep: 1 -prob:1 -servCount:8070
findProduct()
findProduct_reply
<<GaAcqStep>> -rep: 5 -prob:1 -servCount:1258
findItemsRandomByIdProduct()
findItemsRandomByIdProduct_reply
<<GaAcqStep>> -rep: 1 -prob:1 -servCount:4830
findItemRandom()
findItemRandom_reply

(a) UML+MARTE Sequence diagram



Container-Web-Service
<<GaExecHost>> - utilzation: 0.908
Container-Gateway
Container-Products-Server
<<GaExecHost>> - utilzation: 0.10
Container-Categories-Server
<<GaExecHost>> - utilzation: 0.05
Container-Items-Server
<<GaExecHost>> - utilzation: 0.983

(b) UML+MARTE Deployment diagram

Fig. 8: An example of MARTE Annotations

	Web (*Response Time* = 1.615 s)		Warehouse (*Response Time* = 53.332 ms)	
Refactoring Action	*New Response Time*	*Improvement* (%)	*New Response Time*	*Improvement* (%)
Move operation	1.402 s	13.34	50.347 ms	5.04
Increasing device capability	1.281 s	20.73	53.332 ms	0

TABLE I: Performance improvements due to refactoring actions

2 Docker container). After the application of this action, a new UML+MARTE model is obtained and transformed to the corresponding performance model. Thereafter, as a result of its performance evaluation, the PaF occurrence results to be removed. The response time of the Web scenario, in this case, has been improved by 13.34%, whereas the one of the Warehouse scenario has been improved by 5.04%.

By executing an antipattern detection on the refactored model, the PaF and OLB performance antipatterns, previously detected on *Web Service*, still occur as expected. In order to further improve the response time of the Web scenario, we have increased the capabilities of the *Container-Web-Service* device, so that the demand of the *getHome()* operation decreases. As a result of this refactoring, also shown in Table I, the response time of the involved scenario has been improved of 20.73% upon a 4.94% demand reduction. In this case, since the touched device is not involved in the Warehouse scenario, the latter does not experience any performance improvement.

Hence, all the above actions represent the set of refactoring options (i.e., the green box on the right-hand side of Fig. 1) that the joint usage of JTL and PADRE has allowed to identify for removing the performance problems currently occurring in the system.

VI. Related Work

A vast literature exists on performance modeling, performance monitoring, and performance problem identification techniques, as separate research domains. We report on the most significant papers that attempt at merging such domains.

Trubiani et al. [20] provide a systematic process to identify performance issues with runtime data, based on load testing coming from operational profile and application profiling. From the obtained runtime data, performance antipatterns are detected, aimed at identifying common performance issues and their solutions. Software refactoring is then (manually) applied to solve identified performance antipatterns.

Apart from technological and implementation aspects, a methodological difference distinguishes our approach from the one in [20], namely: we bring runtime data up to the design level, by annotating an UML model with the MARTE profile, and one of the main advantage of addressing performance issues at design level is to narrow down the search space for possible later empirical performance tuning.

Menascé et al. [21] have proposed the DeSARM approach, whose scope is the derivation of architectural models at runtime; such models can be used in decentralized decision making for architecture-based adaptation in large distributed systems. To this aim, DeSARM is able to identify important architectural characteristics of a running application, such as components, connectors, nodes and communication patterns. DeSARM has been used by Albassam et al. [22] to introduce runtime failure analysis and architectural recovery on the discovered system architecture. However, DeSARM has not been adopted for identifying performance problems, as we do in this paper.

Petriu et al. [23] deal with the automated generation of performance models from UML-MARTE architectural models, and the propagation of performance analysis results back to the latter. The main difference with our work is that this approach fully work at the model level, and it does not consider the availability of a running software system from which runtime information can be extracted for a more realistic identification of performance problems.

Logs obtained from monitoring the running software system are exploited by Vögele et al. [24] to automatically extract workload specifications for load testing and performance models parameterization. However, [24] is limited to the parameterization of performance models, whilst our approach provides support for the interpretation of analysis results carried out by performance antipattern detection and their possible solutions.

To the best of our knowledge, our approach is the first one that combines system monitoring, traceability links between runtime data and software model, and automated problem identification and solution based on performance antipatterns.

VII. Conclusion and Future Work

In this paper we introduced an approach that aims at supporting the identification and solution of performance problems on a running system by means of automation in: extracting runtime information, annotating design models with performance data, and detecting and solving performance antipatterns.

The application of the proposed approach to the E-Shopper case study allows outlining some considerations. As a premise, the code instrumentation and the utilization of industrial standard technologies, such as Zipkin and Elasticsearch, make the case study adequate to demonstrate the applicability of our approach. Furthermore, the definition and the use of traceability models results to be a key point for automating the exploitation of runtime information for sake of performance analysis and improvement of the system design. In fact, in absence of a collection of correspondences between monitored data and the system design, the extrapolation of a large amount of data from a running application and the annotation of a software model would be very expensive.

We conceived and implemented the approach to be extensible. In this paper, we defined a dedicated metamodel that represents the logs format as represented by the used monitoring infrastructure. However, runtime information can be of different types (eg., simulation or executable models, logs/traces, states or configurations of the system, test models, dynamic information or runtime measures), it can have different formats (eg., textual, binary, datasets) and can be collected by means of various mechanisms (e.g., simulation, monitoring, execution, debugging, profiling, verification). As an enhancement, a generic runtime metamodel that deals with different runtime information can be specified and integrated to the approach.

As a further extension, also the adoption of different modeling languages can be supported by the approach and by the used tools. The designer can specify the correspondences between proper languages; JTL is indeed able to deal with any Ecore artifact conforms to EMF. Moreover, we envisage possibly redefining the model annotation step and the antipattern detection rules; PADRE indeed provides the designer with interfaces to write the proper notation-specific rules.

Finally, as illustrated in Figure 1, the suggested refactoring actions resulting by the application of our approach obviously refer to the architectural design level. As a future work, we intend to close the gap between design and runtime level by introducing automation in the propagation of these actions down to the runtime level (see the System Refactoring dashed box of Figure 1). We will relay on MDE techniques, like the traceability ones in JTL, for such closing step.

References

[1] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] H. Bruneliere, R. Eramo, A. Gomez, V. Besnard, J. M. Bruel, M. Gogolla, A. Kastner, and A. Rutle, "Model-Driven Engineering for Design-Runtime Interaction in Complex Systems: Scientific Challenges and Roadmap - Report on the MDE@DeRun 2018 workshop," in *Proc. of STAF Collocated Workshops*, 2018.

[3] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall, "PerformanceHat: augmenting source code with runtime performance traces in the IDE," in *Proc. of ICSE Companion*, 2018, pp. 41–44.

[4] W. Afzal, H. Brunelière, D. Di Ruscio, A. Sadovykh, S. Mazzini, E. Cariou, D. Truscan, J. Cabot, A. Gómez, J. Gorroñogoitia, L. Pomante, and P. Smrz, "The megam@rt2 ECSEL project: Megamodelling at runtime - scalable model-based framework for continuous development and runtime validation of complex systems," *MICPRO*, vol. 61, pp. 86–95, 2018.

[5] C. M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *FOSE Workshop, ICSE*, 2007, pp. 171–187.

[6] V. Cortellessa, "Performance antipatterns: State-of-art and future perspectives," in *EPEW Proc.*, 2013, pp. 1–6.

[7] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "JTL: A bidirectional and change propagating transformation language," in *SLE Proc.*, 2010, pp. 183–202.

[8] D. Arcelli, V. Cortellessa, and D. Di Pompeo, "Performance-driven software model refactoring," *IST Journal*, vol. 95, pp. 366–397, 2018.

[9] "Unified modeling language," OMG, 2015, version 2.5. [Online]. Available: http://www.omg.org/spec/UML/2.5/

[10] "A UML profile for MARTE: modeling and analysis of real-time embedded systems," OMG, 2008. [Online]. Available: http://www.omg.org/omgmarte/

[11] V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-Based Software Performance Analysis*. Springer, 2011.

[12] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *ICLP*, 1988, pp. 1070–1080.

[13] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV system for knowledge representation and reasoning," *TOCL*, vol. 7, no. 3, pp. 499–562, 2006.

[14] R. Eramo, A. Pierantonio, and M. Tucci, "Improved traceability for bidirectional model transformations," in *Proc. of MDETools workshop, MODELS*, vol. 2245, 2018, pp. 306–315.

[15] V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," *SOSYM*, vol. 13, no. 1, pp. 391–432, 2014.

[16] Kolovos, Dimitris and Rose, Louis and Paige, Richard and Garcıa-Domınguez, Antonio, *The EPSILON book*. Structure, 2010.

[17] R. F. Paige, N. Drivalos, D. S. Kolovos, K. J. Fernandes, C. Power, G. K. Olsen, and S. Zschaler, "Rigorous identification and encoding of trace-links in model-driven engineering," *SOSYM*, vol. 10, no. 4, pp. 469–487, 2011.

[18] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *SOSYM*, vol. 9, no. 4, pp. 529–565, 2010.

[19] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Mar. 1984.

[20] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche, "Exploiting load testing and profiling for performance antipattern detection," *IST Journal*, vol. 95, pp. 329–345, 2018.

[21] J. Porter, D. A. Menascé, and H. Gomaa, "Desarm: A decentralized mechanism for discovering software architecture models at runtime in distributed systems," in *Proc. of Models@run.time workshop, MODELS*, vol. 1742, 2016, pp. 43–51.

[22] H. Gomaa and E. Albassam, "Run-time software architectural models for adaptation, recovery and evolution," in *Proc. of Models@run.time workshop, MODELS*, vol. 2019, 2017, pp. 193–200.

[23] T. Altamimi, M. H. Zargari, and D. C. Petriu, "Performance analysis roundtrip: Automatic generation of performance models and results feedback using cross-model trace links," in *Proc. of CASCON*, 2016, pp. 208–217.

[24] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar, "WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction - a model-driven approach for session-based

application systems," *SOSYM*, vol. 17, no. 2, pp. 443–477, 2018.