Softw Syst Model (2014) 13:391–432
DOI 10.1007/s10270-012-0246-z

REGULAR PAPER

An approach for modeling and detecting software performance antipatterns based on first-order logics

Vittorio Cortellessa · Antinisca Di Marco · Catia Trubiani

Received: 11 April 2011 / Revised: 22 December 2011 / Accepted: 24 March 2012 / Published online: 21 April 2012

Abstract The problem of interpreting the results of performance analysis is quite critical in the software performance domain. Mean values, variances and probability distributions are hard to interpret for providing feedback to software architects. Instead, what architects expect are solutions to performance problems, possibly in the form of architectural alternatives (e.g. split a software component in two components and re-deploy one of them). In a software performance engineering process, the path from analysis results to software design or implementation alternatives is still based on the skills and experience of analysts. In this paper, we propose an approach for the generation of feedback based on performance antipatterns. In particular, we focus on the representation and detection of antipatterns. To this goal, we model performance antipatterns as logical predicates and we build an engine, based on such predicates, aimed at detecting performance antipatterns in an XML representation of the software system. Finally, we show the approach at work on a case study.

Keywords Software performance antipatterns · Performance analysis · Software architectures

1 Introduction

Since more than a decade the problem of modeling and analyzing software performance from the beginning of the lifecycle has been tackled with several approaches. The generation of performance models from software artifacts has gained a core role in this domain, and automation has emerged as a key factor to address some major problems in software development, such as shorter time to market and the lack of specific skills to build performance models.

Figure 1 schematically represents the typical steps of a complete performance modeling and analysis process. In the figure, rounded boxes represent operational steps whereas square boxes represent input/output data. Arrows numbered from 1 through 4 represent the typical forward path from an annotated software architectural model (i.e. a software model enriched with performance-related information) through the production of performance indices of interest.

While in the forward path, well-founded approaches have been introduced to automate all steps [5,7,12,46], there is a clear lack of automation in the backward path that brings the analysis results back to the software architectural model (i.e. arrow 6 in Fig. 1).

The backward path is represented in the figure as a macro-step of results interpretation and feedback generation that in the following we simply call *feedback*. In this step, the performance indices obtained from the model solution, typically represented by average values and/or distribution functions, are interpreted to detect performance problems, if any. A performance problem originates from a set of unfulfilled requirement(s) (such as the estimated response time of a service is higher than the required one). If all the requirements are satisfied, then the feedback obviously suggests no changes.

Once performance problems are found, with a certain accuracy, in the model, solutions can be applied to remove them. Typical solutions consist in design alternatives that

Communicated by Dr. Sebastien Gerard.

V. Cortellessa · A. Di Marco · C. Trubiani (✉)
Dipartimento di Informatica, Università dell'Aquila,
Via Vetoio Coppito (AQ), 67010 L'Aquila, Italy
e-mail: catia.trubiani@univaq.it

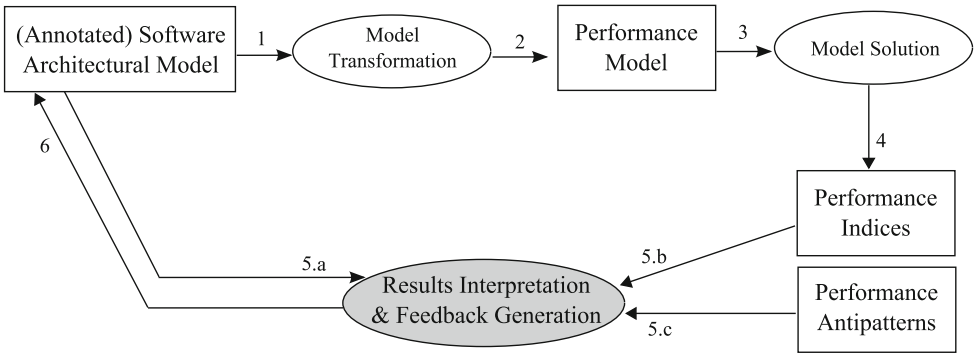


Fig. 1 Software performance modeling and analysis process

modify the original software architectural model to achieve better performance.[1]

As shown in Fig. 1, the (annotated) software architectural model (label 5.a) and the performance indices (label 5.b) are inputs to the feedback step that searches for problems in the model and provides feedback (label 6) to the software architectural model in the form of design alternatives that remove the performance problems found.

This type of search may be quite complex and should be smartly driven towards the problematic areas of the architectural model. The complexity of this step stems from several factors:

(i) the complexity of the software architectural models. The origin of performance problems appear only looking at the several model elements described in different views of a system (such as static structure, behaviour, etc.) requiring that the characteristics among views need to be cross-checked.
(ii) the joint examination of performance indices. A single performance index (e.g. the utilization of a service center) could be not enough to localize the critical parts of an architectural model, a performance problem emerges only if other indices (e.g. the throughput of a neighbor service center) are analyzed.
(iii) the granularity of the performance analysis. The indices can be estimated at different levels of granularity (e.g. the response time can be evaluated at the level of a cpu device, or at the level of a system service that spans on different devices).

Therefore, the need of guidance in this searching process is clear. Strategies to drive the search can rely on information that may depend on several factors such as the adopted software and performance modeling notations[2], the application domain (e.g. real time systems), the environmental constraints.

In this context, we think that the most promising elements that can drive this search are *performance antipatterns* [37] (label 5.c in Fig. 1). Antipatterns basically represent typical patterns (independent from the adopted model notation, the application domain, etc.) that, if occurring in a model, may induce performance problems. An antipattern definition includes: the bad practices specification in terms of model elements (i.e. the problem), and the actions to take to solve the problem (i.e. the solution). Hence we consider that the feedback step benefits from performance antipatterns by focusing the searching process on detecting antipatterns that have well-known solutions.

In this paper, we present an approach to make performance analysis results usable at the software architectural model level. In particular, we tackle the problem of using performance antipatterns to provide an implementation of the Results Interpretation and Feedback Generation steps from Fig. 1. This paper is an extension of [13], with the following main contributions: (i) the formalization of all the 12 antipatterns we examine; (ii) a case study that clearly shows the beneficial effects of using antipatterns in the backward path of the software performance lifecycle.

Moreover, since the publication of [13] the authors have gained more experience using antipatterns in concrete modeling languages (e.g. [11,44]). This experience led us to consider additional issues that are included in the formalization presented in this paper, making it more mature. For example, the Concurrent Processing Systems antipattern logic-based representation has been refined in comparison to [13] by introducing a higher level of detail in the analysis of hardware resources (see more details in Sect. 4.1.3).

[1] We can compare this scenario of performance-driven software design assistance with the work of a physician: observing a sick patient (the model), studying the symptoms (unsatisfactory values of performance indices), making a diagnosis (a software problem), prescribing a therapy (design alternatives).

[2] Modeling is of crucial relevance since different models and relative notations describe the software system highlighting different features of it. Different modeling notations can be adopted to describe both the software architectural models (e.g. UML [3], Automata [23], Process Algebras [30], etc.) and performance ones (e.g. Queueing Networks [26], Simulation Models [6], etc.).

The paper is organized as follows. Section 2 provides more technical details on the implementation we propose for the feedback step of the software performance process. Sections 3 and 4 present the informal representation and the logical predicates-based formalization of antipatterns, respectively. Section 5 describes the engine, based on the specified predicates, which detects the performance antipatterns and provides suitable guidelines to remove them. In Sect. 6, we propose a case study as a proof of concept of our approach, while in Sect. 7 we discuss the open issues of the proposed approach. Section 8 presents related work, and finally Sect. 9 concludes the paper providing remarks and future works.

2 Antipatterns-based approach

Up to now, in literature, performance antipatterns have only been defined in natural language [37]. Hence, the core question addressed in this work is: how to represent a performance antipattern to support its automatic detection? To this aim, we introduce a modeling notation-independent representation of performance antipatterns.

Figure 2 sketches the implementation we provide for the feedback step. Here, a *software architectural model* is an abstract view of the software system. It is composed of complementary types of models providing several system information from different perspectives: (i) the *static* perspective showing the software resources; (ii) the *dynamic* perspective showing the behavior of the system; (iii) the *deployment* perspective showing the hardware resources, etc. [38]. For the sake of analysis, models must be annotated with performance-related information such as the system workload, the service demands, the hardware characteristics, etc.[3] We refer to these enriched models as *(annotated)* ones.

Performance indices [24] (see Fig. 2) refer to extra functional properties of the software system: (i) response time (i.e. the time interval between a user request of a service and the response of the system); (ii) utilization (i.e. the ratio of busy time of a resource in a time slot); (iii) throughput (i.e. the rate at which requests can be handled by a system).

To make performance antipatterns processable (that means detectable and solvable) by a machine, we execute a preliminary *Modeling* step during which we specify antipatterns as logical predicates. Such predicates define conditions on software architectural model elements (e.g. number of interactions among components, resource utilization, etc.) needed to detect antipatterns. We organized these architectural model elements in an *XML Schema* reported in Appendix A.

Starting from an annotated software architectural model (label 5.a) and its performance indices (label 5.b), we execute an *Extracting* step during which the extractor engine generates: (i) an *XML representation of the Architectural Model* conforming to the XML Schema that contains all and only the software architectural model information we need to detect performance antipatterns; (ii) a set of *Antipatterns Boundaries* that drive the interpretation of performance analysis results, since they define thresholds that will be compared with the predicted performance values to support the detection of the performance critical elements.

The *Detection* step is the operational counterpart of the antipatterns declarative definitions as logical predicates. In fact, it takes as input the XML representation of the software architectural model and the antipatterns boundaries, and it returns a list of *detected performance antipatterns* instances, i.e. the description of the detected problems as well as their solutions (instantiated on the architectural model elements). Such list represents the feedback, since it consists of a set of alternative refactoring actions, suggested to the software designers in the backward path (label 6 of Fig. 2), and aimed at removing the detected antipatterns.

Note that, both the Modeling and the Detection steps of the process (see Fig. 2) are reusable across different modeling languages. In contrast, the Extractor engine needs to understand the language in which the annotated software model and the performance indices are expressed, hence it depends on them. Although performance indices refer to the performance model, the Extractor engine needs to understand both modeling languages (for software and performance) and their mapping. The problem of bridging software and performance models is not trivial, since a large semantic gap between such models may occur [32].

The formalization we propose for performance antipatterns provides only the concepts that the antipatterns require to be automatically detected. This means that in the formalization we do not include concepts that are useless for the antipatterns representation. Modeling languages may have a broader scope that covers a large and diverse set of application domains, hence many concepts are not useful for our aims. For example, in UML [3] the state machines are used for modeling discrete behavior through finite state-transition systems and many concepts (e.g. *connectionPointReference*, *port*, *ProtocolConformance*, etc.) are ignored in our formalization.

Moreover, the formalization is not coupled with the detection engine. Currently, our engine has been implemented in Java, but other types of engine can be implemented on the basis of the formalization we provide.

In this paper, the decision on which antipatterns to remove is taken by the software designer, who will manually execute

[3] It is necessary to add annotations to parameterize the performance models. This information either comes from the requirements the system undergoes, or is estimated from performance experts or is observed from similar existing systems. In particular, the workload specification is part of the user requirements and indicates the amount of the expected requests to the software system.

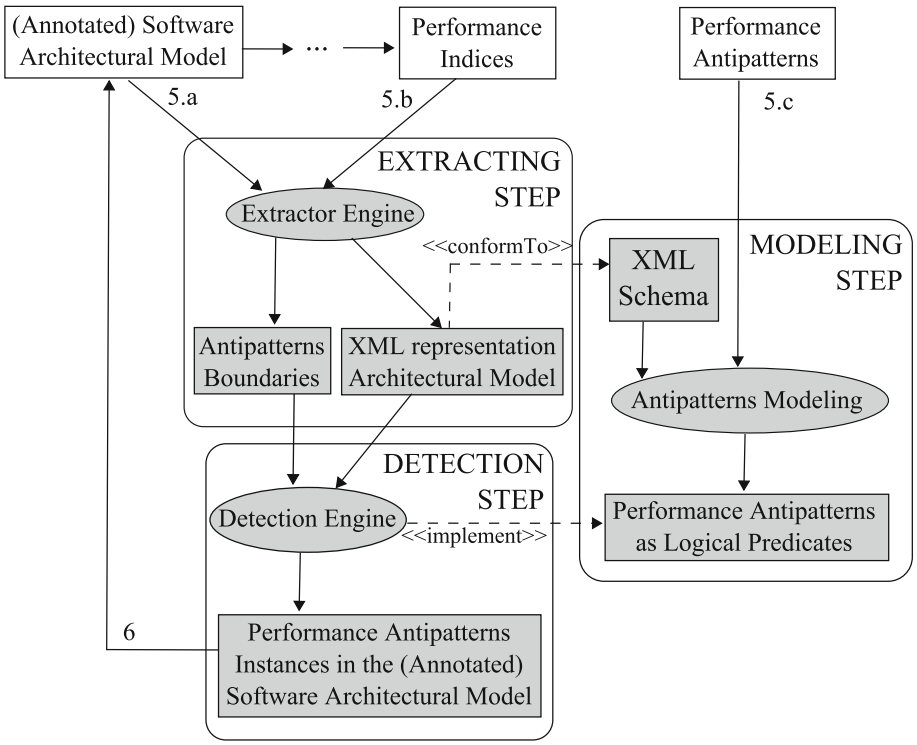


Fig. 2 Results Interpretation and Feedback Generation step

the suggested refactoring action. Indeed this final task could be automated only under certain assumptions, because the application of antipattern solutions can be restricted by several constraints. Examples of such constraints may refer to: (i) the usage of legacy components that cannot be split and re-deployed whereas the antipattern solution suggests such actions; (ii) the development process is subject to budget limitations that do not allow to adopt an antipattern solution due to its extremely high cost. Many other examples can be provided of constraints that (implicitly or explicitly) may affect the antipattern solution activity (see more details in Sect. 7). Due to these complex situations, we prefer to delegate the selection of the antipatterns to be removed to the software designers that can take into account some of the constraints the system must satisfy.

As a concluding remark, we want to point out that the solution of one or more antipatterns does not guarantee a priori performance improvements, because the entire process is based on heuristic evaluations (e.g. antipatterns boundaries, see Fig. 2) and decisions. However, an antipattern-based refactoring action is usually a correctness-preserving transformation that improves the quality of the software. For example, the interaction between two components might be refactored to improve performance by sending fewer messages with more data per message. This transformation does not alter the semantics of the application, but it may improve the overall performance.

Hence, the forward path has to be taken again, starting from the updated software architectural model and ending up with new performance indices that might confirm if performance problems are actually removed. Only after this validation the software lifecycle can proceed, in fact if the validation is unsuccessful then the backward path has to run again and the whole process is repeated.[4]

3 Performance antipatterns definition

Patterns are common solutions to problems that occur in many different contexts [20]. Design patterns capture expert knowledge about "best practices" in software design in a form that allows knowledge to be reused and applied in the design of many different types of software.

Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems [9]. They are known as antipatterns because their use may produce negative consequences. Antipatterns document common mistakes (i.e. "bad practices") made during software development, as well as solutions of the problems deriving from these mistakes.

Performance Antipatterns define bad practices that induce performance problems, and their solutions. In this paper, we refer only to antipatterns that are independent on the notation used to represent software and performance models. This apparent restriction allows to focus on the most common

[4] In a perfect analogy with what a (good) physician does to check if the prescribed therapy works.

antipatterns that can occur in any model, because they are not related to any specific modeling notation. For a similar reason, performance antipatterns of interest do not have to be specific to any application domain.

The main source of performance antipatterns is the work done across years by Smith and Williams that have ultimately defined a number of 14 notation- and domain-independent antipatterns [37]. They describe settings where sub-optimal performance design choices have been made. Some other papers introduce antipatterns that may occur throughout different technologies, but they are not as general as the ones defined by Smith and Williams (see Sect. 8).

Table 1 lists all the antipatterns we consider. Each row of Table 1 represents a specific antipattern, and it characterized by three fields: antipattern name, problem and solution textual descriptions. From the original list of 14 antipatterns defined by Smith and Williams in [37], only two antipatterns have not been considered for the following reason: (1) the *Falling Dominoes* antipattern refers, besides the performance problems, to some reliability and fault tolerance issues, so it is at the moment out of our interest; (2) the *Unnecessary Processing* antipattern deals with the semantics of a software application because it considers the relevance of the application code, i.e. an information not usually included in software architectural models.

This list of performance antipatterns has been here enriched with an additional attribute. As shown in the leftmost part of Table 1, we have partitioned antipatterns in two different categories: antipatterns detectable by single values of performance indices (such as mean, max or min values), named as *single-value* Performance Antipatterns, and antipatterns requiring the trend (or evolution) of the performance indices during the time to capture the performance problems in the software system, named as *multiple-values* Performance Antipatterns. The mean, max or min values are not sufficient to define the latter category of antipatterns, unless these values refer to several observation time frames. Due to these characteristics, the performance indices needed to detect such antipatterns must be obtained via system simulation or monitoring.

An example of a single-value antipattern is the *Circuitous Treasure Hunt* (see Table 1), since it occurs when a large amount of processing is required to perform a task and the mean value is able to capture the problem. An example of a multiple-values antipattern is the *Traffic Jam* (see Table 1), since it occurs when processing time increases as the system is used and the trend of values is required to catch such behavior. A graphical representation of these two antipatterns is provided in Figs. 3 and 4, respectively, where both antipatterns are visualized in UML-like notation for a quick comprehension.

Note that the graphical representation of the antipatterns provides our interpretation of the informal definition reported in Table 1: different formalizations of antipatterns can be originated by laying on different interpretations of their textual specification [37].

Figure 3 provides a graphical representation of the Circuitous Treasure Hunt antipattern.

The upper side of Fig. 3 describes the system properties of a *Software Architectural Model S* with a *Circuitous Treasure Hunt problem*: (a) *Static View* there is a software entity instance, e.g. S_x, retrieving a *lot* of information from a database; (b) *Dynamic View* the software instance S_x generates a *large* number of database calls by performing several queries; (c) *Deployment View* the processing node on which the database is deployed, i.e. $pNode_{DB}$, might reveal a *high* utilization value among its cpu(s) and/or disk(s) devices ($\$utilP_1, \ldots, \$utilP_n$, $\$utilM_1, \ldots, \$utilM_m$). Let us define with $\$maxProcUtil$ and $\$maxMemUtil$ the maximum utilization among cpu(s) and disk(s) devices, respectively, hence at least one of these values overcomes a threshold boundary. Furthermore, a database transaction usually requires a higher utilization of disk devices instead of cpu ones, hence $\$maxMemUtil$ is expected to be larger than $\$maxProcUtil$. The occurrence of such properties leads to assess that the software entity *Database* originates an instance of the Circuitous Treasure Hunt antipattern in the Software Architectural Model S (see Table 1).

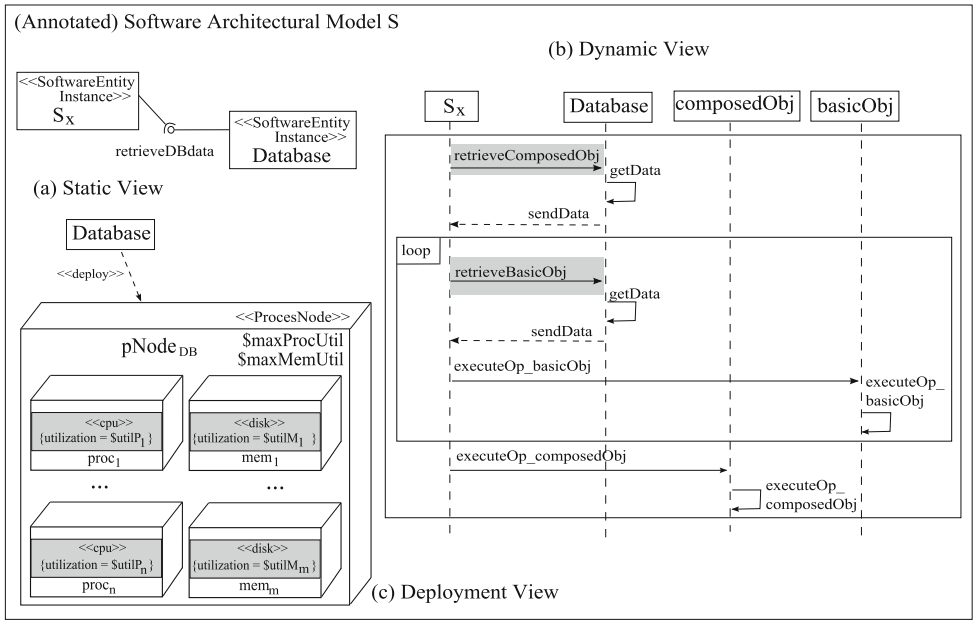
The lower side of Fig. 3 contains the design changes that can be applied according to the *Circuitous Treasure Hunt solution*. The following refactoring action is represented: (a) the database must be restructured to reduce the number of database calls and to retrieve the needed information with fewer database transactions. As consequences of the previous action, if the information is retrieved with a smarter organization of the database, then the utilization of hardware devices is expected to improve in the *Software Architectural Model* S', i.e. $\$maxProcUtil'$ and particularly $\$maxMemUtil'$.

Figure 4 provides a graphical representation of the Traffic Jam antipattern.

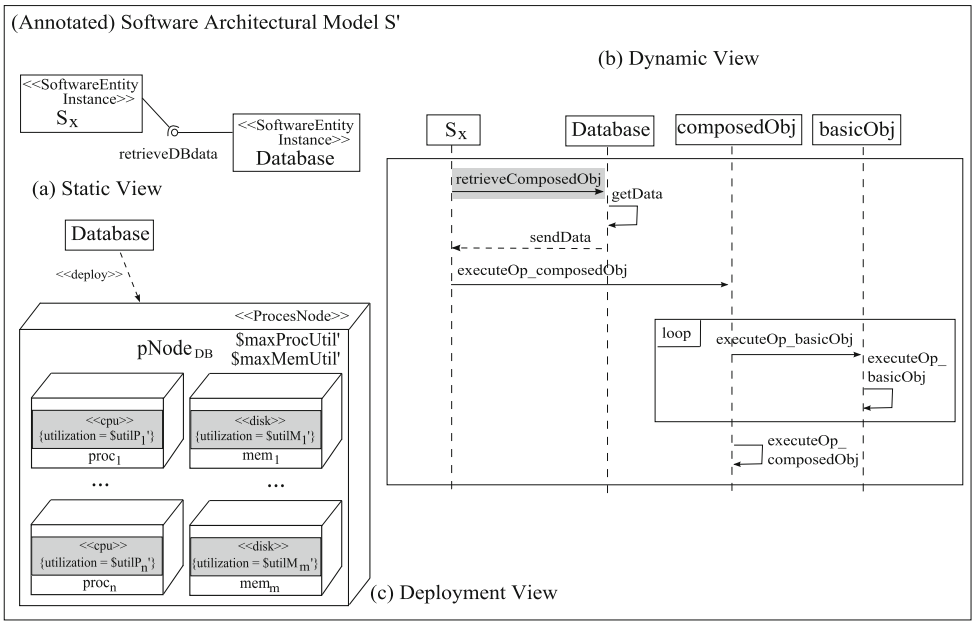
The upper side of Fig. 4 describes the system properties of a *Software Architectural Model S* with a *Traffic Jam problem*: (a) *Static View*, there is a software entity instance S offering an operation op; (b) the (predicted) response time of the operation op shows "*a wide variability in response time which persists long*" [34]. It means that there is a time interval showing a wide variability, e.g. the response time of the operation op in the time interval k ($\$RT(op, k)$) is *much* lower than the (predicted) response time of the operation op in the subsequent time interval $k+1$ ($\$RT(op, k+1)$). The variability persists in the interval from $k+1$ to n, hence the (predicted) response time of the operation op in the time interval $k+1$ ($\$RT(op, k+1)$) is almost equal to the (predicted) response time of the operation op in the time interval n ($\$RT(op, n)$). The occurrence of such properties leads to assess that the software instance S originates an instance

Table 1 Performance antipatterns [37]

	Name	Problem	Solution
Single-value	Circuitous Treasure Hunt	Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look, performance will suffer	Refactor the design to provide alternative access paths that do not require a circuitous treasure hunt (or to reduce the cost of each look)
	Blob (or god class/component)	Occurs when a single class or component either (1) performs all of the work of an application or (2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance	Refactor the design to distribute intelligence uniformly over the application's top-level classes, and to keep related data and behavior together
	Unbalanced Processing		
	Concurrent Processing Systems	Occurs when processing cannot make use of available processors	Restructure software or change scheduling algorithms to enable concurrent execution
	"Pipe and Filter" Architectures	Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput	Break large filters into more stages and combine very small ones to reduce overhead
	Extensive Processing	Occurs when extensive processing in general impedes overall response time	Move extensive processing so that it does not impede high traffic or more important work
	Empty Semi Trucks	Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both	The batching performance pattern combines items into messages to make better use of available bandwidth
	Tower of Babel	Occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML	The fast path performance pattern identifies paths that should be streamlined. Minimize the conversion, parsing, and translation on those paths
	One-Lane Bridge	Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g. when accessing a database)	To alleviate the congestion, use the shared resources principle to minimize conflicts
	Excessive Dynamic Allocation	Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance	(1) Recycle objects (via an object pool) rather than creating new ones each time they are needed. (2) Use the Flyweight pattern to eliminate the need to create new objects
Multiple-values	The Ramp	Occurs when processing time increases as the system is used	Select algorithms or data structures based on maximum size or use algorithms that adapt to the size
	Traffic Jam	Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared	Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load
	More is Less	Occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources	Quantify the thresholds where thrashing occurs (using models or measurements) and determine if the architecture can meet its performance goals while staying below the thresholds



"Circuitous Treasure Hunt" problem



"Circuitous Treasure Hunt" solution

Fig. 3 A graphical representation of the Circuitous Treasure Hunt antipattern

of the Traffic Jam antipattern in the Software Architectural Model S (see Table 1).

The lower side of Fig. 4 contains the design changes that can be applied according to the *Traffic Jam solution*. It is difficult to solve such antipattern at the architectural level because it is not trivial to identify the original cause of the backlog (see Table 1). However, it is possible to increase the processing node rate ($\$processRate'$) on which S is deployed. In fact, such refactoring action is meant to manifest the resolution of the antipattern at the architectural level. As consequences of the previous action, in the *Software Architectural Model S′* the response time of the operation *op* is expected to increase in a slower way, i.e. $\$RT(op, k+1)'$.

4 Performance antipatterns as logical predicates

In this section, we formalize the representation of performance antipatterns as logical predicates. The basic idea is that an antipattern identifies unwanted software and/or hardware properties, thus an antipattern can be formulated as a (maybe complex) predicate on the software architectural model elements.

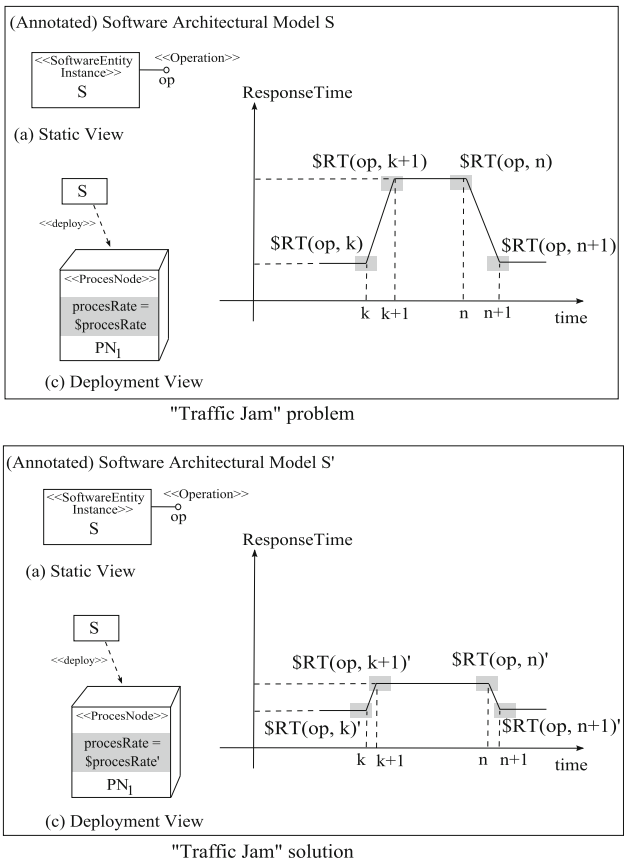


Fig. 4 A graphical representation of the Traffic Jam antipattern

The presentation of the logical predicates is organized into two groups, the single-value (see Sect. 4.1) and the multiple-values (see Sect. 4.2) antipatterns.

A section is dedicated to an antipattern and is organized as follows. From the informal representation of the *problem* (as reported in Table 1), a set of *basic predicates* (BP_i) is built, where each BP_i addresses part of the antipattern problem specification. The basic predicates are first described in a semi-formal natural language and then formalized by means of first-order logics. Note that the operands of basic predicates are XML Schema elements (see Appendix A), here denoted with the `typewriter font`.

We organize the software architectural model elements into views, each capturing a different aspect of the system. Similar to the Three-View Model [45], we consider three different views representing three sources of information: the *Static View* that captures the software elements (e.g. classes, components) and the static relationships among them; the *Dynamic View* that represents the interaction (e.g. messages) that occurs between the software entities elements to provide the system functionalities; and finally the *Deployment View* that describes the hardware elements (e.g. processing nodes) and the mapping of the software entities onto the hardware platforms.

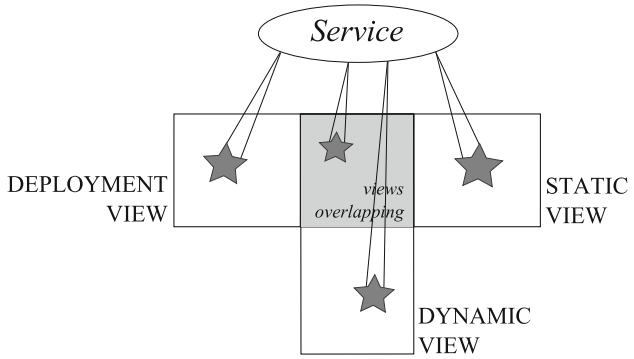


Fig. 5 Bird's-eye look of the Schema and its views

Figure 5 provides a bird's eye look of our XML Schema (it is fully described in Appendix A) that sketches the three views and their intersection[5] in the central and shaded square (i.e. views overlapping). A *Service* is defined at a higher level, because it can be described by means of elements belonging to all the three views.

The benefit of introducing (static, dynamic, deployment) views is that the performance antipattern specification can be partitioned on their basis: the predicate expressing a performance antipattern is in fact the conjunction of sub-predicates, each referring to a different view. However, to specify an antipattern it might not be necessary information coming from all views, because certain antipatterns involve only elements of some views.

In the proposed formalization to capture bad practices, we introduce: (i) a set of *functions* to elaborate system properties represented in the predicates as $F_{funcName}$ (e.g. the number of messages sent by a software entity towards an other one); (ii) a set of *thresholds* to interpret the system features represented in the predicates as $Th_{thresholdName}$ (e.g. the upper bound for the hardware resource utilization). Note that all functions and thresholds are summarized in Appendix C.

4.1 Single-value performance antipatterns

In this section, we report some examples of the *Performance Antipatterns* that can be detected by single values of performance indices (such as mean, max or min values). In particular, we formalize Circuitous Treasure Hunt, Blob, Concurrent Processing Systems, Empty Semi Trucks antipatterns in the following, whereas the remaining ones are reported in Appendix B.

[5] Note that it exists an overlapping among the views (e.g. the elements interacting in the dynamics of a system are also part of the static view).

4.1.1 Circuitous Treasure Hunt

Circuitous Treasure Hunt [37] has the following problem informal definition: *"occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look, performance will suffer"* (see Table 1).

We formalize this sentence with three basic predicates: the BP_1 predicate whose elements belong to the Static and the Dynamic Views; the BP_2 and BP_3 predicates whose elements belong to the Deployment View.

BP_1 There are two `SoftwareEntityInstances`, e.g. swE_x and swE_y, such that: (*i*) they are both involved in a `Service` S; (*ii*) the instance playing the `senderRole` (e.g. swE_x), sends an excessive number of `Messages` to the one,playing the `receiverRole` (e.g. swE_y); (*iii*) the receiver is a database (as captured by the `isDB` attribute). To formalize such interpretation we use the $F_{numDBmsgs}$ function that provides the number of messages sent by swE_x to swE_y in the `Service` S. The property of sending an excessive number of messages can be checked by comparing the output value of the $F_{numDBmsgs}$ function with a threshold $Th_{maxDBmsgs}$:

$$swE_y.isDB = true \tag{1}$$

$$F_{numDBmsgs}(swE_x, swE_y, S) \geq Th_{maxDBmsgs} \tag{2}$$

BP_2 The `ProcesNode` P_{swE_y} on which the software instance swE_y is deployed has a heavy computation. That is, the `Utilization` of hardware entities belonging to the `ProcesNode` P_{swE_y} exceed a certain threshold $Th_{maxHwUtil}$. For the formalization of this characteristic, we recall that the $F_{maxHwUtil}$ function with the 'all' option returns the maximum `Utilization` among all the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{swE_y}, all) \geq Th_{maxHwUtil} \tag{3}$$

BP_3 Since, in general, a database access utilizes more disk than cpu, we require that the maximum disk(s) `Utilization` is larger than the maximum cpu(s) `utilization` of the `ProcesNode` P_{swE_y}:

$$F_{maxHwUtil}(P_{swE_y}, disk) > F_{maxHwUtil}(P_{swE_y}, cpu) \tag{4}$$

Summarizing, the *Circuitous Treasure Hunt* antipattern occurs when the following composed predicate is true:

$$\exists swE_x, swE_y \in \mathrm{swE}, S \in \mathbb{S} \mid \boxed{(1) \wedge (2) \wedge (3) \wedge (4)}$$

where swE represents the set of `SoftwareEntityInstances`, and $\mathbb{S}$ represents the set of `Services` in the software system. Each (swE_x, swE_y, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Circuitous Treasure Hunt antipattern.

4.1.2 Blob (or god class/component)

Blob (or "god" class/component) [33] has the following problem informal definition: *"occurs when a single class either 1) performs all of the work of an application or 2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance."* (see Table 1).

We formalize this sentence with four basic predicates: the BP_1 predicate whose elements belong to the Static View; the BP_2 predicate whose elements belong to the Dynamic View; and finally the BP_3 and BP_4 predicates whose elements belong to Deployment View.

BP_1 Two cases can be identified for the occurrence of the blob antipattern.

In the first case, there is at least one `SoftwareEntity Instance`, e.g. swE_x, such that it "*performs all of the work of an application*" [33], while relegating other instances to minor and supporting roles. Let us define the function $F_{numClientConnects}$ that counts how many times the software entity instance swE_x is in a `Relationship` with other software entity instances by assuming a `clientRole` for swE_x. The property of performing all the work of an application can be checked by comparing the output value of the $F_{numClientConnects}$ function with a threshold $Th_{maxConnects}$:

$$F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \tag{5}$$

In the second case, there is at least one `Software EntityInstance`, e.g. swE_x, such that it "*holds all of the application's data*" [33]. Let us define the function $F_{numSupplierConnects}$ that counts how many times the software entity instance swE_x is in a `Relationship` with other software entity instances by assuming the `supplierRole` for swE_x. The property of holding all of the application's data can be checked by comparing the output value of the $F_{numSupplierConnects}$ function with a threshold $Th_{maxConnects}$:

$$F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects} \tag{6}$$

BP_2 swE_x performs most of the business logics in the system or holds all the application's data, thus it generates or receives excessive message traffic. Let us define by $F_{numMsgs}$ the function that takes as input a software entity instance with a `senderRole`, a software entity instance with a `receiverRole`, and a `Service` S, and returns the multiplicity of the exchanged `Messages`. The property of excessive message traffic can be checked by comparing the output value of the $F_{numMsgs}$ function with a threshold $Th_{maxMsgs}$ in both

directions:

$$F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \quad (7a)$$
$$F_{numMsgs}(swE_y, swE_x, S) \geq Th_{maxMsgs} \quad (7b)$$

The performance impact of the excessive message traffic can be captured by considering two cases. The first case is the *centralized* one (modeled by the BP_3 predicate), i.e. the blob software entity instance and the surrounding ones are deployed on the same processing node, hence the performance issues due to the excessive load may come out by evaluating the utilization of such processing node. The second case is the *distributed* one (modeled by the BP_4 predicate), i.e. the Blob software entity instance and the surrounding ones are deployed on different processing nodes, hence the performance issues due to the excessive message traffic may come out by evaluating the utilization of the network links.

BP_3 The `ProcesNode` P_{xy} on which the software entity instances swE_x and swE_y are deployed shows heavy computation. That is, the `utilization` of a hardware entity of the `ProcesNode` P_{xy} exceeds a certain threshold $Th_{maxHwUtil}$. For the formalization of this characteristic, we use the $F_{maxHwUtil}$ function that has two input parameters: the processing node, and the type of `HardwareEntity`, i.e. 'cpu', 'disk', or 'all' to denote no distinction between them. In this case, the $F_{maxHwUtil}$ function is used to determine the maximum `Utilization` among 'all' the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \quad (8)$$

BP_4 The `ProcesNode` P_{swE_x} on which the software entity instance swE_x is deployed, shows a high utilization of the network connection towards the `ProcesNode` P_{swE_y} on which the software entity instance swE_y is deployed. Let us define by $F_{maxNetUtil}$ the function that provides the maximum value of the `usedBandwidth` overall the network links joining the processing nodes P_{swE_x} and P_{swE_y}. We must check if such value is higher than a threshold $Th_{maxNetUtil}$:

$$F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil} \quad (9)$$

Summarizing, the *Blob (or "god" class/component)* antipattern occurs when the following composed predicate is true:

$$\exists swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid$$
$$\boxed{((5) \vee (6)) \wedge ((7a) \vee (7b)) \wedge ((8) \vee (9))}$$

where sw$\mathbb{E}$ represents the `SoftwareEntityInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (swE_x, swE_y, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Blob antipattern.

4.1.3 Concurrent Processing Systems

Concurrent Processing Systems [35] have the following problem informal definition: *"occurs when processing cannot make use of available processors"* (see Table 1).

We formalize this sentence with three basic predicates: the BP_1, BP_2, BP_3 predicates whose elements belong to the Deployment View. In the following, we denote with $\mathbb{P}$ the set of the `ProcesNode` instances in the system.

BP_1 There is at least one `ProcesNode` in $\mathbb{P}$, e.g. P_x, having a large `QueueLength`. Let us define by F_{maxQL} the function providing the maximum `QueueLength` among all the hardware entities of the processing node. The first condition for the antipattern occurrence is that the value obtained from F_{maxQL} is larger than a threshold Th_{maxQL}:

$$F_{maxQL}(P_x) \geq Th_{maxQL} \quad (10)$$

BP_2 P_x has a heavy computation. This means that the utilizations of some hardware entities in P_x (i.e. cpu, disk) exceed predefined limits. We use the already defined $F_{maxHwUtil}$ to identify the highest `utilization` of cpu(s) and disk(s) in P_x, and then we compare such utilizations to the $Th_{maxCpuUtil}$ and $Th_{maxDiskUtil}$ thresholds:

$$F_{maxHwUtil}(P_x, cpu) \geq Th_{maxCpuUtil} \quad (11a)$$
$$F_{maxHwUtil}(P_x, disk) \geq Th_{maxDiskUtil} \quad (11b)$$

BP_3 The processing nodes are not used in a well-balanced way, as there is at least another instance of `ProcesNode` in $\mathbb{P}$, e.g. P_y, whose `Utilization` of the hardware entities, differentiated according to their type (i.e. cpu, disk), is smaller than the one in P_x. In particular, two new thresholds, i.e. $Th_{minCpuUtil}$ and $Th_{minDiskUtil}$, are introduced:

$$F_{maxHwUtil}(P_y, cpu) < Th_{minCpuUtil} \quad (12a)$$
$$F_{maxHwUtil}(P_y, disk) < Th_{minDiskUtil} \quad (12b)$$

Summarizing, the *Concurrent Processing Systems* antipattern occurs when the following composed predicate is true:

$$\exists P_x, P_y \in \mathbb{P} \mid$$
$$\boxed{(10) \wedge [((11a) \wedge (12a)) \vee ((11b) \wedge ((12b)))]}$$

where $\mathbb{P}$ represents the set of all the `ProcesNodes` in the software system. Each (P_x, P_y) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Concurrent Processing Systems antipattern.

4.1.4 Empty Semi Trucks

Empty Semi Trucks [37] have the following problem informal definition: *"occurs when an excessive number of requests is required to perform a task. It may be due to*

inefficient use of available bandwidth, an inefficient interface, or both" (see Table 1).

We formalize this sentence with three basic predicates: the BP_1 predicate whose elements belong to the Dynamic View; the BP_2 and BP_3 predicates whose elements belong to the Deployment View.

BP_1 There is at least one `SoftwareEntityInstance` swE_x that exchanges an excessive number of `Messages` with remote software entities. Let us define by $F_{numRemMsgs}$ the function that calculates the number of remote messages sent by swE_x in a `Service` S:

$$F_{numRemMsgs}(swE_x, S) \geq Th_{maxRemMsgs} \tag{13}$$

BP_2 The inefficient use of available bandwidth means that the `SoftwareEntityInstance` swE_x sends a high number of messages without optimizing the network `capacity`. Hence, the `ProcesNode` P_{swE_x}, on which the software entity instance swE_x is deployed, reveals an utilization of the network lower than the threshold $Th_{minNetUtil}$. We focus on the `NetworkLink`(s) that connect P_{swE_x} to the whole system, i.e. the ones having P_{swE_x} as their `EndNode`. Since we are interested to the network links on which the software instance swE_x generates traffic, we restrict the whole set of network links to the ones on which the interactions of the software instance swE_x with other communicating entities take place:

$$F_{maxNetUtil}(P_{swE_x}, swE_x) < Th_{minNetUtil} \tag{14}$$

BP_3 The inefficient use of interface means that the software instance swE_x communicates with a certain number of remote instances, all deployed on the same remote processing node. Let us define by $F_{numRemInst}$ the function that provides the maximum number of remote instances with which swE_x communicates in the service S. The antipattern can occur when this function returns a value higher or equal than a threshold $Th_{maxRemInst}$:

$$F_{numRemInst}(swE_x, S) \geq Th_{maxRemInst} \tag{15}$$

Summarizing, the *Empty Semi Trucks* antipattern occurs when the following composed predicate is true:

$$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(13) \wedge ((14) \vee (15))}$$

where sw$\mathbb{E}$ represents the `SoftwareEntityInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (swE_x, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents an Empty Semi Trucks antipattern.

4.2 Multiple-values performance antipatterns

In this section, we report one example of the *Performance Antipatterns* that can be detected requiring the trend (or evolution) of the performance indices during the time (i.e. multiple values) to capture the performance problems induced in the software system. In particular, we formalize the Traffic Jam antipattern in the following, whereas the remaining ones are reported in Appendix B.

4.2.1 Traffic Jam

Traffic Jam [34] has the following problem informal definition: *"occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared"* (see Table 1).

We formalize this sentence with three basic predicates: the BP_1, BP_2, and BP_3 predicates whose elements belong to the Static View.

BP_1 There is at least one `OperationInstance` OpI that has a quite stable value of its response time along different observation time interval up to the kth one. Let us define by F_{RT} the function that returns the mean `ResponseTime` of the `OperationInstance` OpI observed in the interval t. We consider the average response time increase of the operation in $k-1$ consecutive time slots in which no peaks are shown, which means that it is lower than a threshold $Th_{OpRtVar}$:

$$\frac{\sum_{1 \leq t \leq k} |(F_{RT}(OpI, t) - F_{RT}(OpI, t-1))|}{k-1} < Th_{OpRtVar} \tag{16}$$

BP_2 The `OperationInstance` OpI has an increasing value of its response time along the kth observation interval. We consider the average response time increase of the operation in the kth time slot in which a peak is shown, which means that it is higher than a threshold $Th_{OpRtVar}$:

$$|F_{RT}(OpI, k) - F_{RT}(OpI, k-1)| > Th_{OpRtVar} \tag{17}$$

BP_3 The `OperationInstance` OpI has a quite stable value of its response time after the kth observation interval, since the wide variability persists long. Different observation time slots are considered up to the nth observation interval. We consider the average response time increase of the operation in $n-k$ consecutive time slots in which no peaks are shown:

$$\frac{\sum_{k \leq t \leq n} |(F_{RT}(OpI, t) - F_{RT}(OpI, t-1))|}{n-k} < Th_{OpRtVar} \tag{18}$$

Summarizing, the *Traffic Jam* antipattern occurs when the following composed predicate is true:

$$\exists OpI \in \mathbb{O} \mid \boxed{(16) \wedge (17) \wedge (18)}$$

where $\mathbb{O}$ represents the set of all the `OperationInstances` in the software system. Each OpI instance

satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Traffic Jam antipattern.

5 Detection engine

To show the applicability of the defined predicates for the detection of performance antipatterns, we have implemented them as a java rule-engine application. The detection engine uses the Java API for XML processing that is Document Object Model (DOM) [1]; it is a cross-platform and language-independent convention for representing and interacting with objects in XML documents.

Such engine parses any XML document compliant with our Schema and returns the detected antipatterns (that we call antipattern instances) on the software architectural model. Whenever an antipattern instance is detected, the corresponding textual descriptions of the problem and the solution (see Table 1) are automatically tailored to the case by replacing roles with specific elements (see Sect. 6).

For example, the Concurrent Processing Systems antipattern *occurs when processing cannot make use of available processors* (see Table 1).

Our detection engine is meant to match the three conditions under which the *Concurrent Processing Systems* antipattern occurs (see Sect. 4.1.3). The first condition consists in looking for a processing node that has the maximum queue length larger or equal to a threshold; the second condition consists in looking for an unbalanced cpu utilization, and finally the third condition consists in looking for an unbalanced disk utilization.

The detection engine returns as output the specific model elements that participate in the occurrence of the antipattern, and the device (i.e. cpu or disk) that causes the unbalanced utilization. For example, the output produced for our case study is: *processing cannot make use of the processors* libraryNode *and* webServerNode*, caused by unbalanced disk utilization* (see Sect. 6). Such output is a more structured representation of the textual solution reported in Table 1.

6 Case study

In this section, we apply the antipatterns-based approach to an Electronic Commerce System (ECS) case study. With this example, we aim at demonstrating the validity of our approach by illustrating how the predicates introduced in Sect. 4 can be used to detect performance antipatterns.

Figure 6 customizes the approach of Fig. 1. The software system is modeled with UML [3] and annotated with the MARTE profile [4] that provides all the information we need

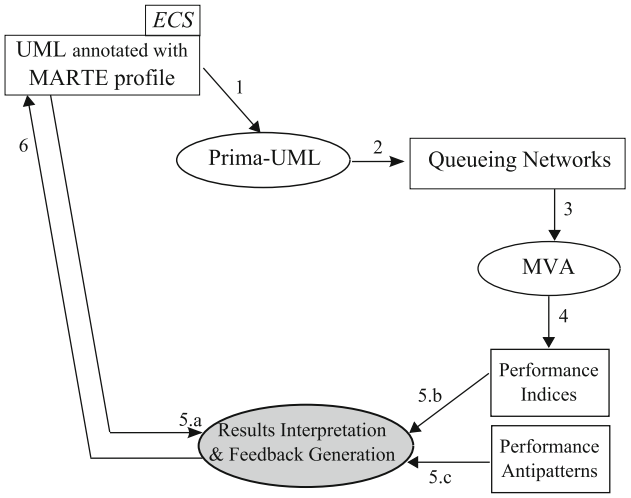


Fig. 6 ECS case study: customized software performance process

for reasoning on performance issues.[6] The transformation from the software architectural model to the performance model is performed with PRIMA-UML, i.e. a methodology that generates a Queueing Network (QN) model from UML diagrams [15]. Once the QN model is derived, classical QN solution techniques based on well-known methodologies can be applied to solve the model, such as Mean Value Analysis (MVA) or simulation [24]. The performance model is analyzed to obtain the performance indices of interest such as response time, utilization, throughput. The *results interpretation and feedback generation* are implemented as presented in Sect. 2 (see Fig. 2). We recall that it is composed of three operational steps: (i) the *modeling* step where performance antipatterns are defined as predicates modeling specific properties of the UML models and MARTE profile annotations (see Sect. 4); (ii) the *extracting* step where antipatterns boundaries and XML representation are generated from the UML model and the performance indices (see Sect. 6.3); (iii) the *detection* step where the java rule-engine provides the critical elements in architectural models representing the source of performance problems as well as a set of guidelines suggested to the designer that actually decide the ones to apply (see Sect. 6.4).

6.1 (Annotated) software architectural model

Figure 7 shows an overview of the ECS software system. It is a web-based system that manages business data: customers browse catalogues and make selections of items that need to

[6] MARTE (Modeling and Analysis of Real-Time and Embedded Systems) profile provides facilities to annotate UML models with information required to perform performance analysis (e.g. workload to the system, service demands, hardware characteristics, etc.).

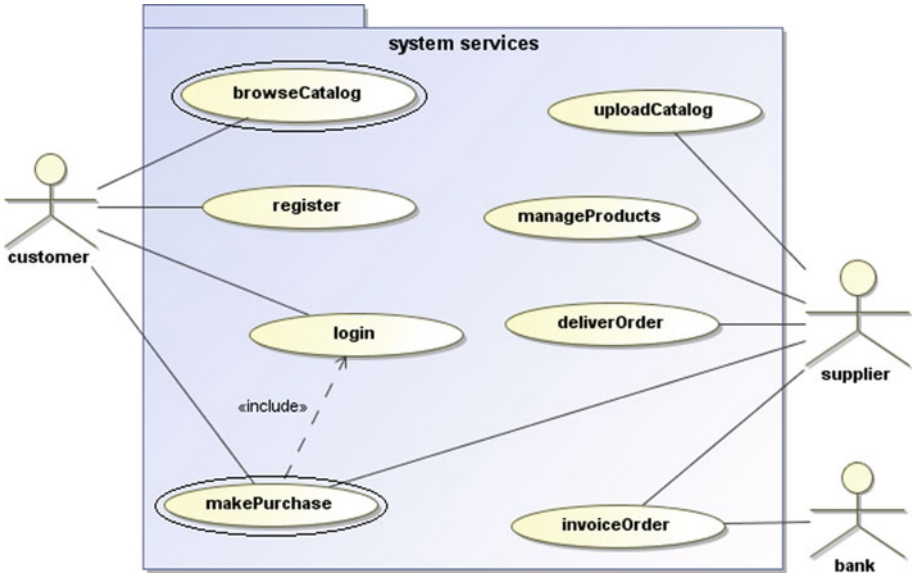


Fig. 7 UML Use Case Diagram of ECS

be purchased; at the same time, suppliers can upload their catalogues, change the prices and the availability of products etc.

The performance-critical services of the ECS are *browseCatalog* and *makePurchase*, i.e. the ones highlighted by a double circle in Fig. 7. The former can be performance critical because it is required by a large number of (registered and not registered) customers, whereas the latter can be performance critical because it requires several database accesses that can drop the system performance.

Performance requirements have been defined on the response time of these two ECS services under a workload of 200 requests/second. The requirements are defined as follows: the *browseCatalog* service has to be offered in 1.2 s, whereas the *makePurchase* one in 2 s.

In Fig. 8, we report an excerpt of the ECS annotated software architectural model used in the analysis of *browseCatalog* and *makePurchase* services. In particular, the UML component diagram in Fig. 8a describes the software components and their interconnections, whereas the UML deployment diagram of Fig. 8b shows the deployment of the software components on the hardware platforms. The deployment is annotated with the characteristics of the hardware nodes to specify CPU attributes (*speedFactor* and *schedPolicy*) and network delay (*blockT*).

6.2 Performance model

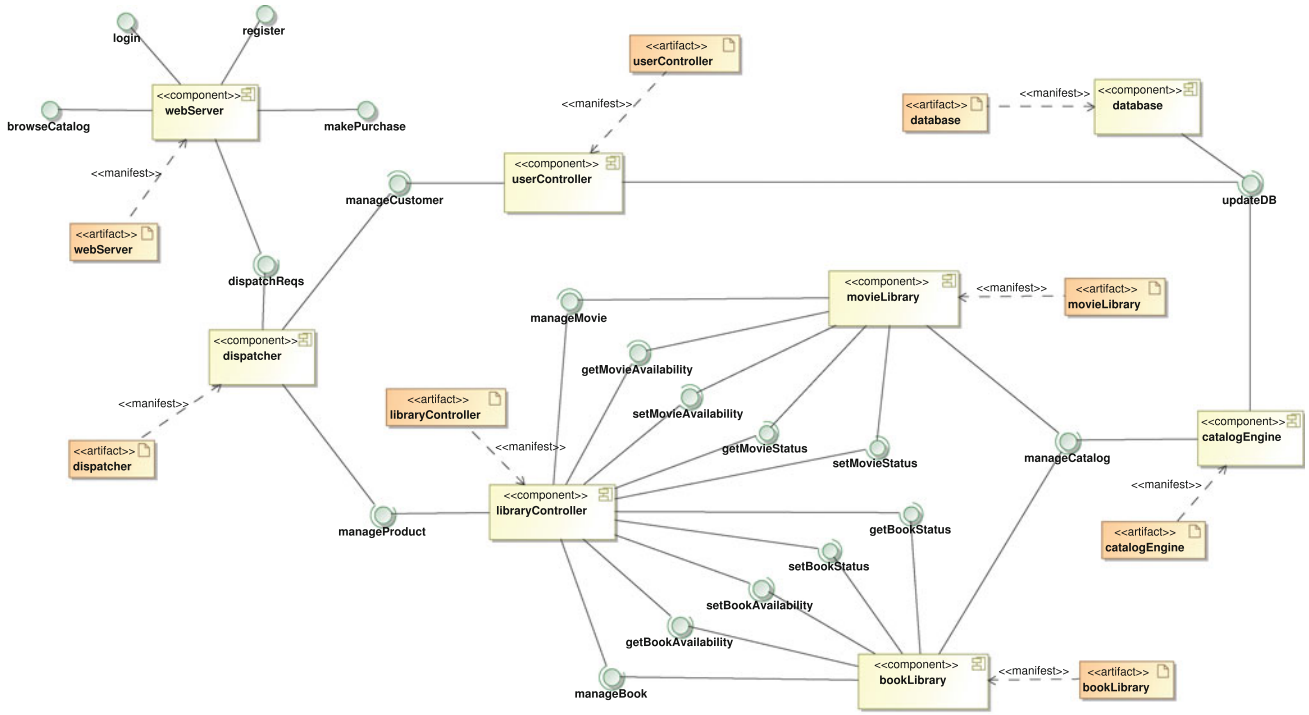
Figure 9 shows the Queueing Network model produced with Prima-UML for the ECS system. Queueing Network models have been widely applied as system performance models to represent and analyze resource sharing systems [28,26,25,43].

A QN model is a collection of interacting *service centers* representing system resources and a set of *jobs* representing the users sharing the resources [28]. Its informal representation is a direct graph whose nodes are service centers and their connections are represented by the graph edges. Jobs go through the graph's edge set on the basis of the behavior of customers' service requests. Service centers may be of two types: queueing and delay. Customers at a *queueing* center compete for the use of the server, thus the time spent by a customer at a queueing center has two components: the time spent waiting and the time spent receiving service. Customers at a *delay* center are allocated to their own server, so there is no competition for service. Delay centers are useful in any situation in which it is necessary to impose some known average delay.

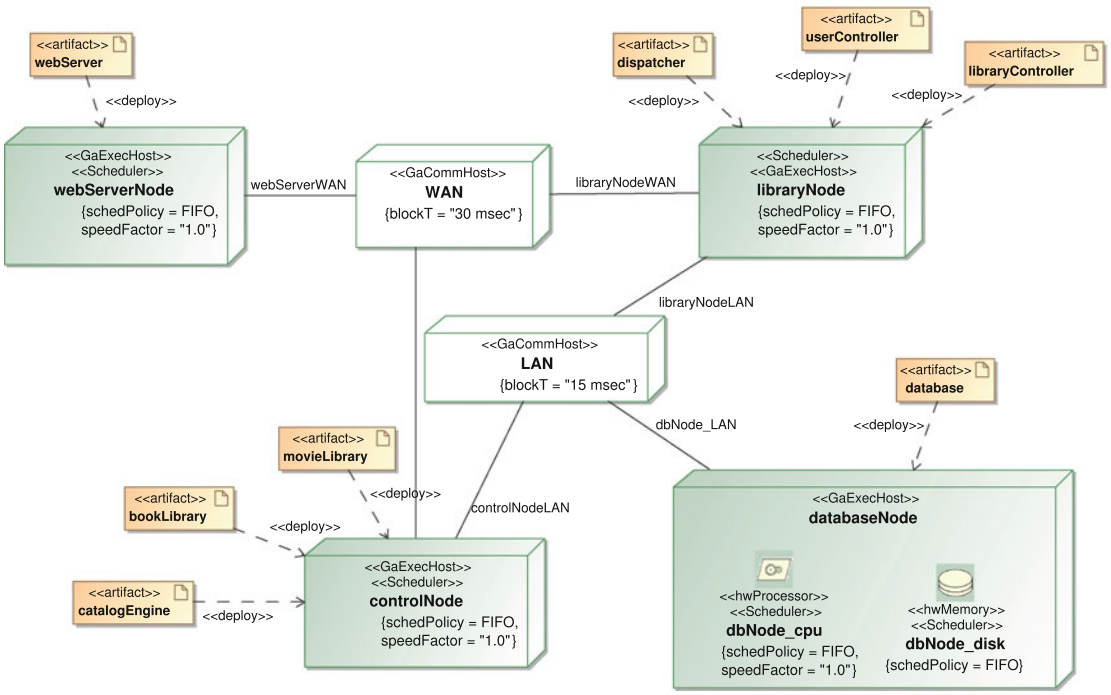
The performance analysis executed through the QN model can be split into three steps: *definition* that include the definition of service centers, their number, class of customers and topology; *parametrization*, to define model parameters, e.g. arrival processes, service rate and number of customers; *evaluation*, to obtain a quantitative description of the modeled system, by computing a set of performance indices such as resource utilization, system throughput and customer response time.

The definition of the Queueing Network model for the ECS case study (see Fig. 9) includes: (i) a set of queueing centers (e.g. *webServerNode*, *libraryNode*, etc.) representing the hardware resources of the system, a set of delay centers (e.g. *wan1*, *wan2*, etc.) representing the network communication delays; (ii) two classes of jobs, i.e. *browseCatalog* (*class A*, denoted with a star symbol in Fig. 9) invoked with a probability of 99 %, and *makePurchase* (*class B*, denoted with a bullet point in Fig. 9) is invoked with a probability of 1 %.

The parametrization of the Queueing Network model for the ECS case study is extracted by Prima-UML from the UML models of the ECS and for convenience collected in



(a) UML Component Diagram.



(b) UML Deployment Diagram.

Fig. 8 ECS (Annotated) Software Architectural Model

Table 2 where the first column contains the service centers and the second column shows their corresponding service rates for each class of job (i.e. *class A* and *class B*).

The evaluation of the Queueing Network model for the ECS case study is collected in Table 3 that reports the performance-critical services (first column), their response time requirements (second column), and the corresponding value obtained by the prediction analysis (third column).

As it can be noticed, both services have a response time that does not fulfill the required ones: (i) the *browse*

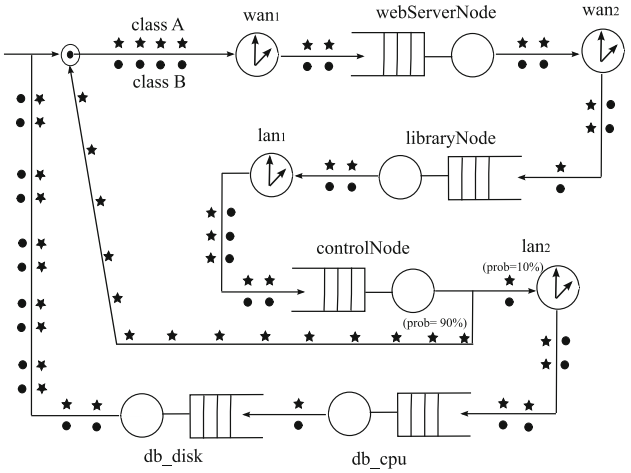


Fig. 9 ECS—Queueing Network model

Table 2 Input parameters for the Queueing Network model in the ECS system

Service center	Input parameters ECS	
	class A (ms)	*class B* (ms)
lan	44	44
wan	208	208
webServerNode	2	4
libraryNode	7	16
controlNode	3	3
db_cpu	15	30
db_disk	30	60

Table 3 Response time requirements for the ECS software architectural model

Requirement	Required value (s)	Predicted value ECS (s)
RT(*browseCatalog*)	1.2	1.5
RT(*makePurchase*)	2	2.77

Catalog service has been predicted of 1.5 s and the requirement is 1.2 s; (ii) the *makePurchase* has been predicted of 2.77 s and the requirement is 2 s. Hence, we apply our approach to detect performance antipatterns.

6.3 Extract step

As first step, the approach joins the Software Architectural Model and the Performance Indices (see Fig. 2) in an XML representation[7] of the ECS case study.

[7] The XML representation of the ECS can be viewed in http://www.di.univaq.it/catia.trubiani/phDthesis/ECS.xml.

As said in Sect. 4, basic predicates contain boundaries that need to be actualized on each specific architectural model.

In our e-commerce system the numerical values are obtained by means of heuristics defined in the following.

The **Blob** antipattern (Sect. 4.1.2) requires four boundaries:

$Th_{maxConnect}$

Definition: it specifies the bound for the number of the components' usage dependencies;

Heuristics: it has been calculated as the average number of usage dependencies among all the components instances of the software architectural model.

$Th_{maxMsgs}$

Definition: it delimits the number of messages sent by a component;

Heuristics: it has been calculated as the average number of messages sent by all the components in a certain service.

$Th_{maxHwUtil}$

Definition: it specifies the maximum utilization value for execution hosts.

Heuristics: it has been calculated as the average utilization among all the execution hosts (see Appendix C, with $\epsilon = 0.25$).

$Th_{maxNetUtil}$

Definition: it specifies the maximum utilization value for communication hosts.

Heuristics: it has been calculated as the average utilization among all the communication hosts (see Appendix C, with $\epsilon = 0.25$).

Note that if architectural constraints are explicitly stated in the requirements (e.g. the average utilization of the hardware machines must not be higher than 0.7), then such requirements can be used to replace the boundaries values (e.g. $Th_{maxHwUtil}$ and $Th_{maxHwUtil}$ should be both set to 0.7).

The **Concurrent Processing Systems** antipattern (Sect. 4.1.3) requires five boundaries:

$Th_{maxQueue}$

Definition: it defines the bound for the number of requests incoming to the node;

Heuristics: it has been calculated as the average number of the queue size among all the nodes' instances of the software architectural model.

$Th_{maxCpuUtil}$, $Th_{minCpuUtil}$

Definition: they are used to specify, respectively, the upper and the lower bound of the processor utilization.

Heuristics: they have been calculated as the average utilization among all the processors (see Appendix C, with $\epsilon = 0.25$).

$Th_{maxDiskUtil}$, $Th_{minDiskUtil}$

Definition: they are used to specify, respectively, the upper and the lower bound of the disk utilization.

Heuristics: they have been calculated as the average utilization among all the disks (see Appendix C, with $\epsilon = 0.25$).

The **Empty Semi Trucks** antipattern (Sect. 4.1.4) requires three boundaries:

$Th_{remMsgs}$

Definition: it delimits the number of remote messages sent by a component.

Heuristics: it has been calculated as the average number of remote messages sent by all the components in a certain service.

$Th_{remInst}$

Definition: it specifies the bound for the number of the remote communicating instances;

Heuristics: it has been calculated as the average number of communicating instances among all the components of the software architectural model.

$Th_{minNetUtil}$

Definition: it represents the lower bound for the network link utilization;

Heuristics: it has been calculated as the average utilization of all the network link instances (see Appendix C, with $\epsilon = 0.25$).

The **Traffic Jam** antipattern (Sect. 4.2.1) requires one boundary:

$Th_{OpRtVar}$

Definition: it delimits the variability in response times of operations.

Heuristics: it has been calculated as the average slope of the response time observed in consecutive time slots for a certain operation.

Similar estimates must be done for all antipatterns. We recall that Appendix C reports all the thresholds we need to evaluate antipatterns boundaries. In particular, bindings for their numerical values are discussed.

Table 4 reports the binding of the performance antipatterns boundaries for the ECS system. Such values allow to enrich the basic predicates with the *performance boundaries* (explained in Sect. 2), thus to proceed with the actual detection.

Table 4 ECS- performance boundaries binding

Antipattern	Parameter	Value
Blob	$Th_{maxConnect}$	4
	$Th_{maxMsgs}$	18
	$Th_{maxHwUtil}$	0.75
	$Th_{maxNetUtil}$	0.85
CPS	$Th_{maxQueue}$	40
	$Th_{maxCpuUtil}$	0.8
	$Th_{maxDiskUtil}$	0.7
	$Th_{minCpuUtil}$	0.3
	$Th_{minDiskUtil}$	0.4
EST	$Th_{remMsgs}$	12
	$Th_{remInst}$	5
	$Th_{minNetUtil}$	0.35
TJ	$Th_{OpRtVar}$	400
…	…	…

6.4 Detect step

The detection of antipatterns is performed by running the detection engine on the XML representation of the ECS software architectural model. The java rule-engine reports the presence of three performance antipatterns: Blob, Concurrent Processing Systems (CPS), and Empty Semi Trucks (EST), and the relative solution guidelines are reported in Table 5.

The **Blob** antipattern is originated by (*lc1*, *bl1*, *browseCatalog*) instances.

In Fig. 10, we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the zones that evidence the Blob antipattern occurrence. Such antipattern is detected in the ECS software architectural model since there is the instance *lc1* of the component *libraryController* such that (see Table 4 and Fig. 10): (a) it has more than *4* usage dependencies towards the instance *bl1* of the component *bookLibrary*; (b) it sends more than *18* messages (not shown in Fig. 10 for sake of space); (c) the component instances (i.e. *lc1* and *bl1*) are deployed on different nodes, and the LAN communication host has an utilization (i.e. 0.92), higher than the threshold value (*0.85*).

The java rule-engine application suggests the following refactoring action: *Refactor the design to keep related data and behavior together: delegate some work from* `lc1` *to* `bl1` (see Table 5).

The **CPS** antipattern is originated by (*libraryNode*, *webServerNode*) instances.

In Fig. 11 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the zones that evidence the CPS antipattern occurrence. Such antipattern is detected in the ECS software architectural model since there are two nodes, i.e. *libraryNode* and *webServerNode* that (see Table 4 and Fig. 11) are not used in a well-balanced way. It means that: (i) the queue size of *libraryNode* (i.e. 50) is higher than the threshold value of *40*; (ii) an unbalanced load among CPUs does not occur, because the maximum utilization of CPUs in *libraryNode* (i.e. 0.82) is higher than *0.8* threshold value, but the maximum utilization of CPUs in *webServerNode* (i.e. 0.42) is not lower than *0.3* threshold value; (iii) an unbalanced load among disks occurs, in fact the maximum utilization of disks in *libraryNode* (i.e. 0.78) is higher than the threshold value of *0.7*, and the maximum utilization of disks in *webServerNode* (i.e. 0.35) is lower than the threshold value of *0.4*.

The java rule-engine application suggests the following refactoring action: *Restructure software or change scheduling algorithms between processors* libraryNode *and* webServerNode (see Table 5).

Table 5 ECS Performance Antipatterns: problem and solution

Antipattern	Problem	Solution
Blob	The libraryController instance *lc1* performs most of the work, it generates excessive message traffic	Refactor the design to keep related data and behavior together: delegate some work from *lc1* to the bookLibrary instance *bl1*
Concurrent processing systems	Processing cannot make use of the processors *libraryNode* and *webServerNode* caused by unbalanced disk utilization	Restructure software or change scheduling algorithms between processors *libraryNode* and *webServerNode*
Empty semi trucks	An excessive number of requests is performed by the userController instance *uc1* to perform the *makePurchase* service	Combine items into messages (for the communication that *uc1* originates) to make better use of available bandwidth

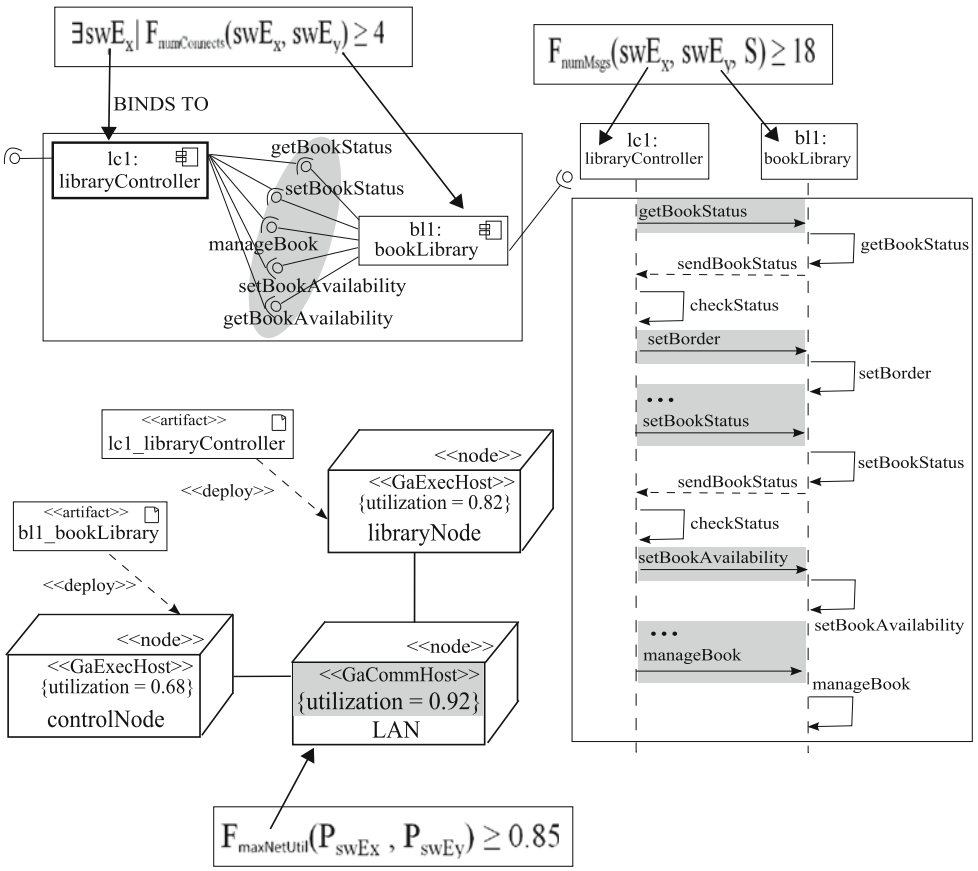


Fig. 10 ECS—the *Blob* antipattern occurrence

The **EST** antipattern is originated by (*uc1*, *makePurchase*) instances.

In Fig. 12 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the zones that evidence the EST antipattern occurrence. Such antipattern occurs since there is the instance *uc1* of the *userController* component such that (see Table 4 and Fig. 12): (a) it sends more than *12* remote messages (not shown in Fig. 12 for sake of space); (b) the component instances are deployed on different nodes, and the communication host has a utilization (i.e. 0.25 in the *wan* instance), lower than *0.35* threshold value; (c) it has more than *5* remote instances (*ce1*, …, *ce8*) of the *catalogEngine* component with which it communicates.

The java rule-engine application suggests the following refactoring action: *Combine items into messages (for the communication that* uc1 *originates) to make better use of available bandwidth* (see Table 5).

6.5 Model refactoring

According to the antipattern solutions proposed in Table 5, we refactor the model and we obtain three new software architectural models, named $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$, where the Blob, the CPS and the EST antipatterns have been solved, respectively (see Fig. 13).

In particular, the *Blob* antipattern is solved by modifying the inner behavior of the *libraryController* software

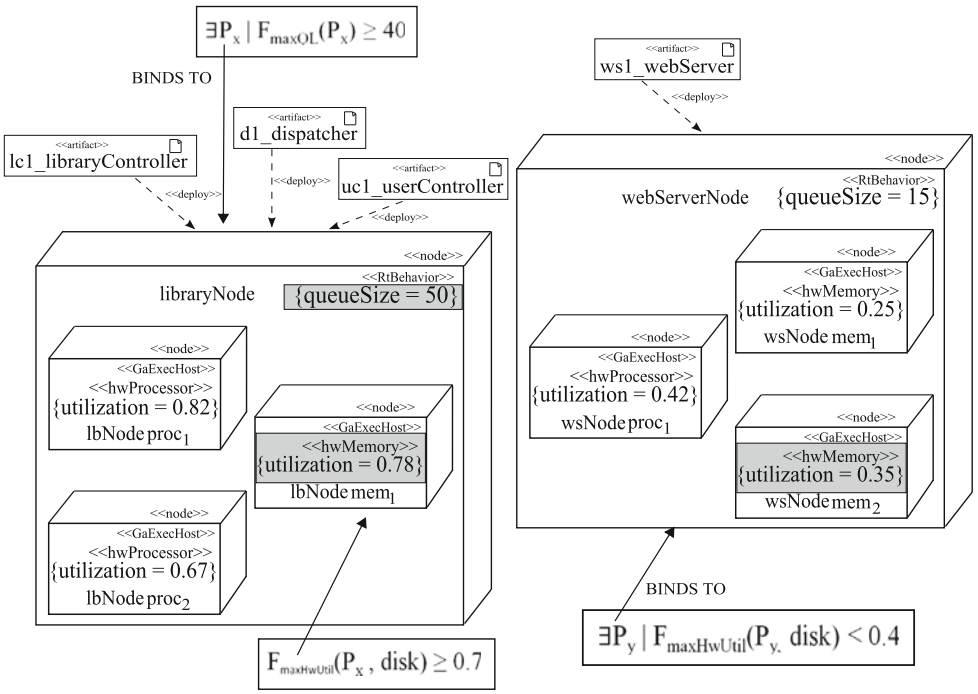


Fig. 11 ECS—the *Concurrent Processing Systems* antipattern occurrence

component, thus it is not anymore the intermediate component for services provided by the *bookLibrary* and *movieLibrary* components.

The *CPS* antipattern is solved by re-deploying the software component *userController* from *libraryNode* to *webServerNode*.

The *EST* antipattern is solved by modifying the inner behavior of the *userController* software component that has less computation in the communication with the *catalogEngine* component in the *makePurchase* service.

$ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$ systems have been analyzed. Input parameters are reported in Table 6 where shaded entries represent the changes induced from the solution of the corresponding antipatterns. For example, in the column $ECS \setminus \{cps\}$, we can notice that the queueing centers *webServerNode* and *libraryNode* have different input values, since the redeployment of the software component *userController* implies to move the load in the involved resources: (i) in case of the job *class A*, the load is estimated of 2 ms and it is moved from *libraryNode* (the initial value of 2 ms in ECS is increased of 2 ms, thus to become 4 ms in $ECS \setminus \{cps\}$) to *webServerNode* (the initial value of 7 ms in ECS is decreased of 2 ms, thus to become 5 ms in $ECS \setminus \{cps\}$); (ii) in case of the job *class B*, the load is estimated of 8 ms and it is moved from *libraryNode* (the initial value of 4 ms in ECS is increased of 8 ms, thus to become 12 ms in $ECS \setminus \{cps\}$) to *webServerNode* (the initial value of 16 ms in ECS is decreased of 8 ms, thus to become 8 ms in $ECS \setminus \{cps\}$).

Table 7 summarizes the performance results obtained by solving the QN models of the new ECS systems, i.e. the systems where design alternatives (as suggested by the antipattern solutions) are applied. Under a workload of 200 requests/second, the response time of the *browseCatalog* service is 1.14, 1.15, and 1.5 s for $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$, respectively. The response time of the *makePurchase* service is 2.18, 1.6, and 2.24 s for $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$, respectively.

The Blob antipattern solution has satisfied the first requirement, but not the second one. The solution of the Concurrent Processing System leads to satisfy both requirements. Finally, the Empty Semi Trucks solution was useless for the first requirement as no improvement was carried out, but it was quite beneficial for the second one, even if both of them were not fulfilled.

Hence, the experimental results highlight that each antipattern has a different impact on performance indices, and in our case study we found that the Concurrent Processing System was the most beneficial one, since all performance requirements were fulfilled after removing it.

6.6 Extending the system modeling

The application of the antipatterns-based approach is not limited (in principle) along the software lifecycle, but it is obvious that an early usage is subject to limited information because the system knowledge improves while the development process progresses. Both the architectural and the performance models can be further detailed with additional knowledge.

The aim of this section is to demonstrate that as far as the system knowledge advances the antipatterns-based approach

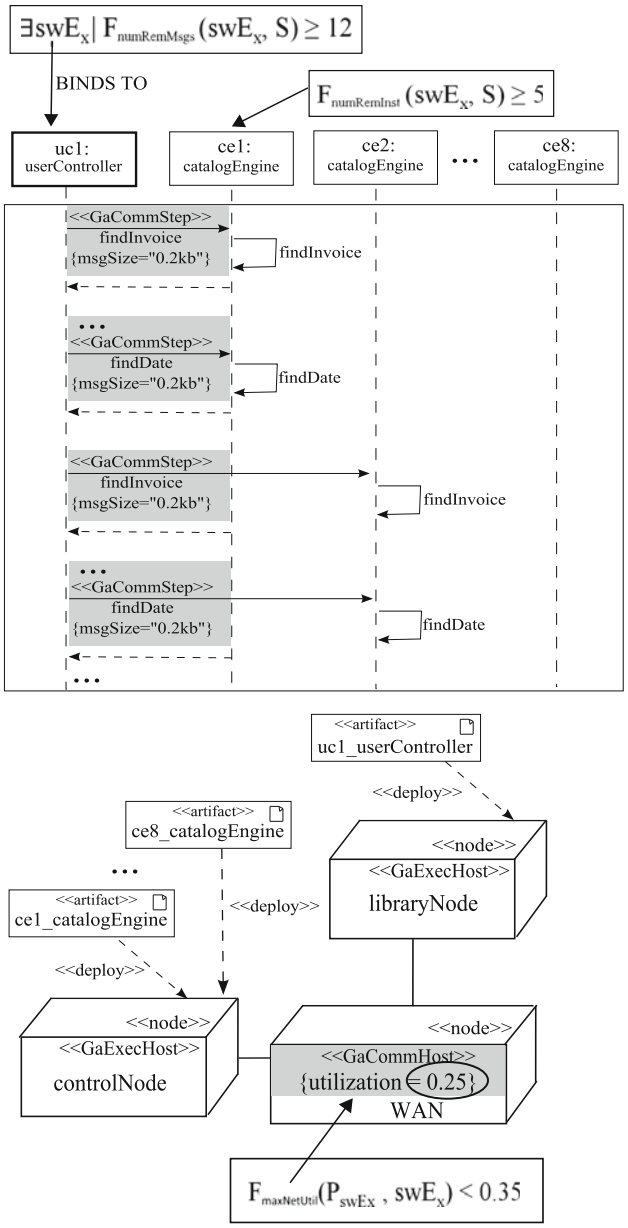


Fig. 12 ECS—the *Empty Semi Trucks* antipattern occurrence

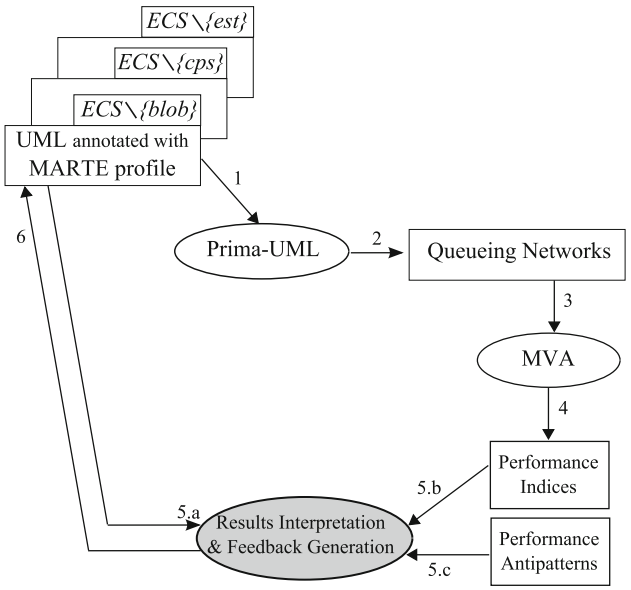


Fig. 13 ECS model refactoring: reiteration of the software performance process

can suggest suitable architectural alternatives for the system under development. We discuss how the modeling of an additional service in the ECS case study (i.e. *uploadCatalog* performed by the suppliers of the system, see Fig. 14), leads the process to provide an extended evaluation of the software system.

Figure 15 shows an excerpt of the annotated ECS software architectural model used in the following analysis. In particular, Fig. 15a and 15b, respectively, reports the UML component and deployment diagrams where dashed boxes contain the architectural elements we added for modeling the *uploadCatalog* service. Note that some model elements are already shown in Fig. 8, since some of them (e.g. *catalogEngine*, *database*) are used by both customer and supplier users.

Figure 16 shows the Queueing Network model produced with Prima-UML for the ECS system, and it is extended with the *uploadCatalog* service. The definition of this Queueing Network model includes: (i) the introduction of queueing centers (e.g. *supplierWebServerNode*) representing the additional hardware resources of the system; (ii) the introduction of one class of jobs, i.e. *uploadCatalog* (*class C*, denoted with a square symbol in Fig. 16).

The parametrization of the Queueing Network model for the ECS case study is extracted by Prima-UML from the UML models of the ECS and, for convenience, summarized in Table 8 where the first column contains the service centers and the second column shows their corresponding service rates for the classes of job.

Note that the Queueing Network model of Fig. 16 is a mixed Queueing Network [26], in fact there are multiple types of job classes: the network is open for jobs of *classC* and closed for *classA* and *classB*. The model of Fig. 16 has been solved by simulation with the Java Modelling Tools (JMT) [10].

Both the architectural and the performance models include additional knowledge that has been inserted in our XML representation, and the detection process has been newly performed. The java rule-engine reports the presence of a new performance antipattern, i.e. the Traffic Jam, as reported in Table 9.

The **Traffic Jam** antipattern is originated by the *uploadCatalog* instance.

Table 6 Input parameters for the Queueing Network model across different software architectural models

Service Center	Input parameters					
	$ECS \setminus \{cps\}$		$ECS \setminus \{est\}$		$ECS \setminus \{blob\}$	
	$classA$ (ms)	$classB$ (ms)	$classA$ (ms)	$classB$ (ms)	$classA$ (ms)	$classB$ (ms)
lan	44	44	44	44	44	44
wan	208	208	208	208	208	208
$webServerNode$	4	12	2	4	2	4
$libraryNode$	5	8	7	12	5	14
$controlNode$	3	3	3	3	3	3
db_cpu	15	30	15	30	15	30
db_disk	30	60	30	60	30	60

Table 7 Response time requirements for $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$ software architectural models

Requirement	Required value (s)	Predicted value		
		$ECS \setminus \{blob\}$ (s)	$ECS \setminus \{cps\}$ (s)	$ECS \setminus \{est\}$ (s)
RT($browseCatalog$)	1.2	1.14	1.15	1.5
RT($makePurchase$)	2	2.18	1.6	2.24

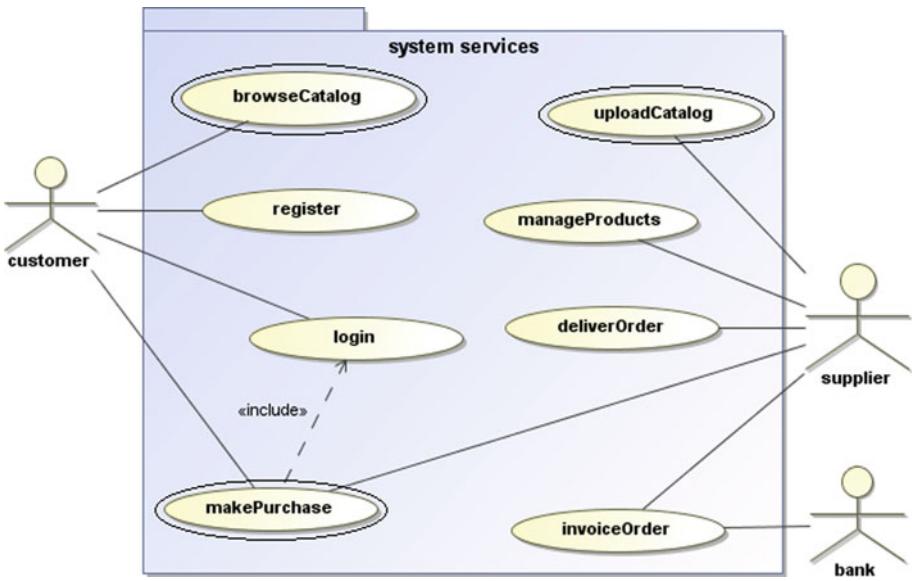
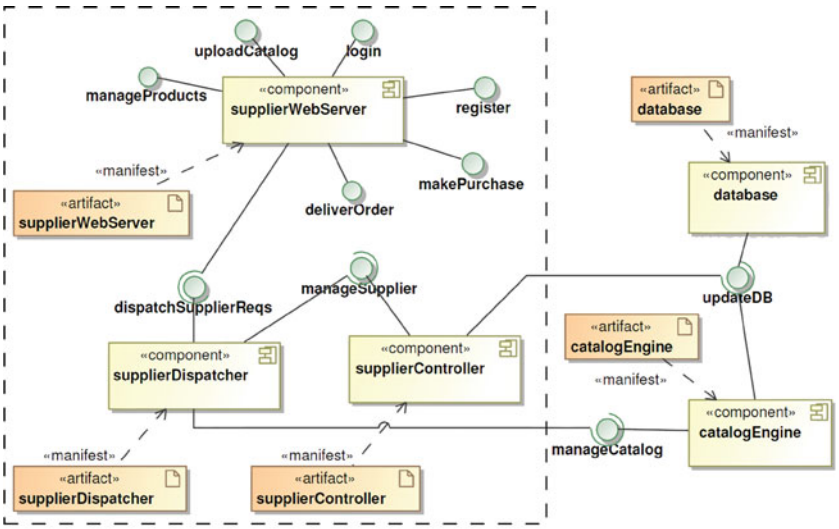


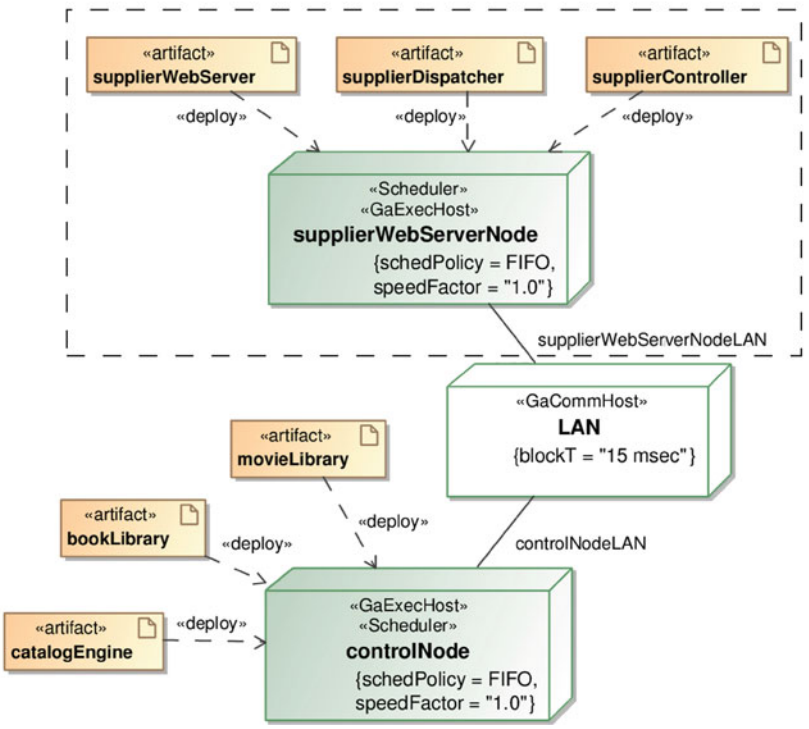
Fig. 14 UML Use Case Diagram of ECS: extending the modeling to the *uploadCatalog* service

Figure 17 illustrates an excerpt of the response time (observed over simulation time) of the *uploadCatalog* service, where we highlight the Traffic Jam antipattern occurrence. On the x axis, the simulation time is reported, and on the y axis the response time of the service is depicted. Single points in the graph (denoted with the $\times$ symbol) represent the *observed values*. For example, the point (1184.93, 184.93) means that the completion a job of *classC* (i.e. the *uploadCatalog* service) has been observed at 1184.93 simulation instant and the response time measured for it is 184.93 ms. Figure 17 additionally shows the trend of the average response time for the *uploadCatalog* service. We obtained this trend by dividing the simulation time in intervals of 1000 ms, and for each interval we calculate the average response time of the observed completions. Hence we draw the *average trend* by considering the calculated average response time as constant in the referring interval, thus to obtain the piecewise linear function, i.e. the solid line of Figure 17. For example, in the simulation time interval [5000, 6000] several jobs of *classC* have been completed and their average response time is 450.16 ms.

By observing the average trend, it is worth to notice that several intervals show the occurrence of the Traffic Jam antipattern, i.e. [0, 5000], [4000, 16000], [12000, 20000],



(a) UML Component Diagram, with the modeling of the *uploadCatalog* service.



(b) UML Deployment Diagram, with the modeling of the *uploadCatalog* service.

Fig. 15 ECS (Annotated) Software Architectural Model, with the modeling of the *uploadCatalog* service

[17000, 26000], [22000, 30000]. For example, in the interval [4000, 16000] we can notice that the operation *uploadCatalog* shows the following features: (a) it has a quite stable value of its response time along different observation time slots up to 11000 ms of simulation time; (b) it has an increasing value of its response time in the intervals [10000, 11000] and [11000, 12000], in fact a peak is shown: RT(uploadCatalog, 10000) = 461.84 ms, and RT(uploadCatalog, 11000) = 951.21 ms by giving raise to a gap of 489.37 ms that is larger than the $Th_{OpRtVar}$ threshold value

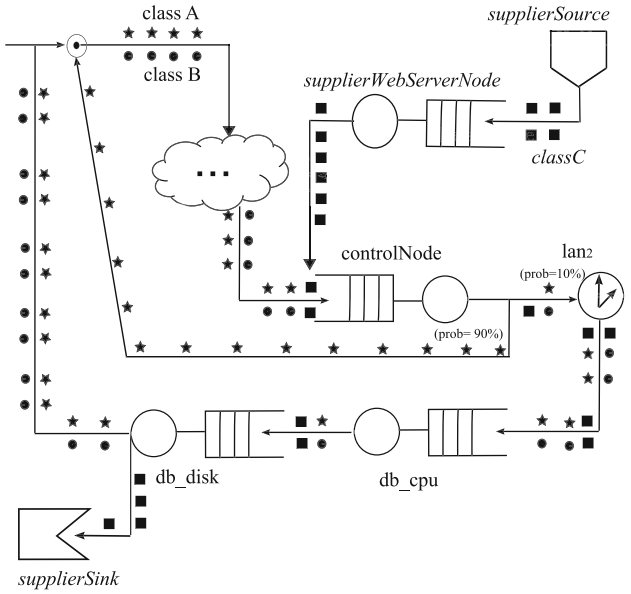


Fig. 16 ECS—Queueing Network model, with the modeling of the *uploadCatalog* service

Table 8 Input parameters for the Queueing Network model in the ECS system

Service center	Input parameters ECS		
	classA (ms)	*classB* (ms)	*classC* (ms)
lan	44	44	44
supplierWebServerNode	–	–	13
controlNode	3	3	3
db_cpu	15	30	40
db_disk	30	60	80

set to 400 ms (see Table 4); (c) it has a quite stable value of its response time after 11000 ms of simulation time, in fact RT(uploadCatalog, 12000) = 306.44 ms.

The java rule-engine application suggests the following refactoring action: *Provide sufficient processing power to handle the worst-case load for the* uploadCatalog *operation instance* (see Table 9).

According to the antipattern solution proposed in Table 9, we refactor the ECS model and we obtain a new software architectural model, named $ECS \setminus \{tj\}$, where the Traffic Jam antipattern has been solved. In particular, it is solved by substituting some hardware devices (i.e. *supplierWebServerNode*, *controlNode*, and *databaseNode*) with faster ones, thus to increase their processing power. Figure 18 illustrates the performance improvement we experimented for the response time of the *uploadCatalog* service, i.e. for both the single observed values and the average trend in the different time slots. The maximum slope is observed from the intervals [14000, 15000] and [15000, 16000] of simulation time where there is a difference of 317.20 ms in the response time of the uploadCatalog service, i.e. lower than the $Th_{OpRtVar}$ threshold value (see Table 4).

7 Discussion

The formalization of antipatterns proposed in this paper is the result of a long process that has produced several formulations before the current one. To study how to formalize them, we have first looked for (simple and complex) examples in literature [9,36]. We have compared the definitions from Smith [37] and the retrieved examples. Then, we have identified the system elements encountered in such study and we have organized them in the XML Schema described in Sect. 4 and fully reported in Appendix A. Thereafter, we have interpreted the antipatterns to provide a notation-independent and a machine-processable definition of them. To reach this goal, we have focused on a first-order logic representation that provides the necessary expressiveness to describe the conditions under which an antipattern occurs.

However, although the used notation is very formal and it can be used to automatize the antipattern detection, we must point out that this formalization reflects our interpretation of the informal textual definition of the considered antipatterns. Indeed several other feasible interpretations of antipatterns can be provided. This unavoidable gap is an open issue in this domain, and certainly requires a wider investigation to consolidate the antipatterns' definition.

Moreover, the formal definition of antipatterns is subject to another degree of uncertainty due to the presence of a certain number of thresholds. They introduce a degree of freedom in the antipattern detection. Such thresholds (i.e. the *Antipatterns Boundaries* of Fig. 2) must be bound to

Table 9 ECS Performance Antipatterns: problem and solution

Antipattern …	Problem …	Solution …
Traffic Jam	The *uploadCatalog* operation instance causes a backlog of jobs that produces wide variability in response time	Provide sufficient processing power to handle the worst-case load for the *uploadCatalog* operation instance

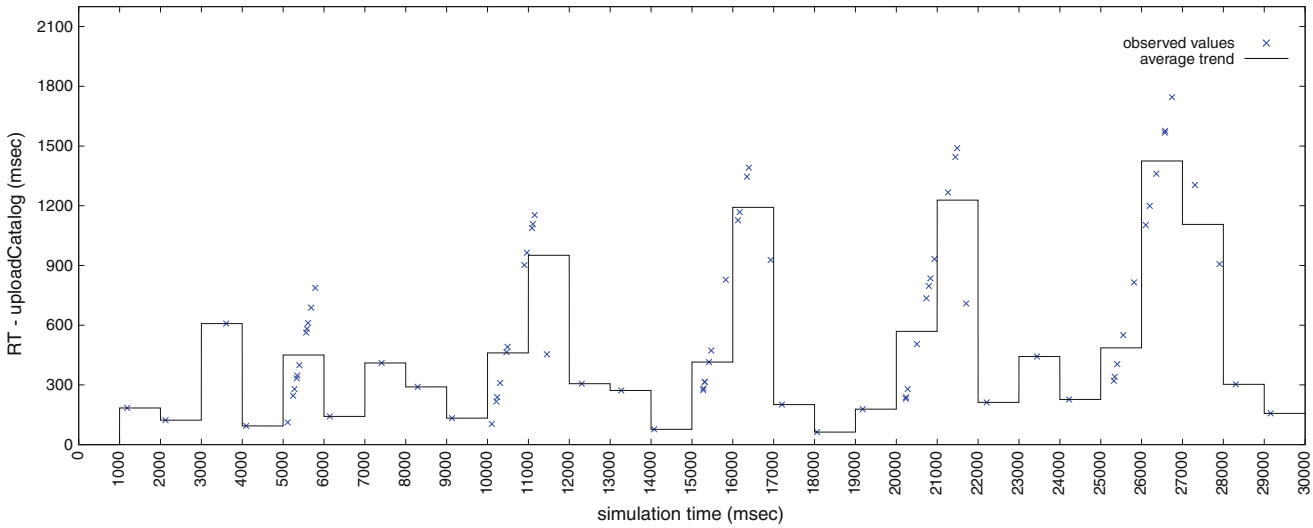


Fig. 17 ECS—the *Traffic Jam* antipattern occurrence

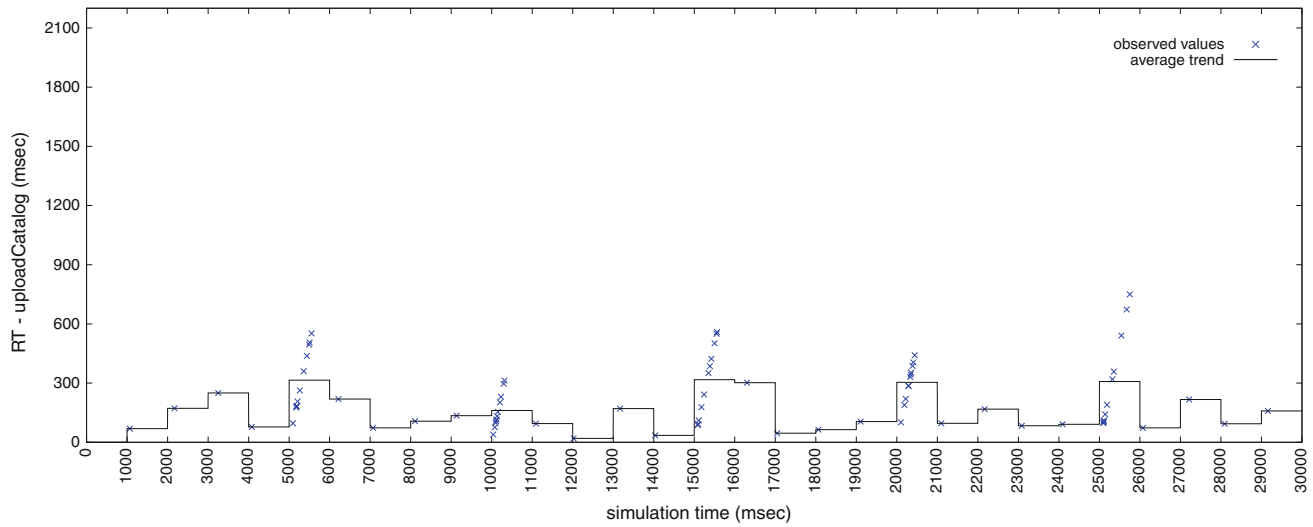


Fig. 18 $ECS \setminus \{tj\}$—performance improvement due to the *Traffic Jam* antipattern solution

concrete numerical values (e.g. hardware entities whose utilization is higher than 0.8 are considered critical ones), and providing guidelines to assign those values is again an open issue. For example, one can either define some heuristics for them (as we did in this paper and reported in Tables in Appendix C), or set them directly on the basis of her experience, or they can be extracted from the system requirements. However, threshold values can be refined as far as more trustable sources of information are available. Additionally, the bounding of some thresholds is intrinsically more difficult than others. For example, both the Ramp and the Traffic Jam antipatterns refer to thresholds representing the maximum feasible slope of the response time (or the throughput) observed in consecutive time slots, hence these values are not easy to assign. Adaptive heuristics can be introduced to iteratively obtain more accurate threshold boundaries. For example, in case of the Ramp and the Traffic Jam antipatterns, such heuristics may exploit historical data (obtained by previous performance analyses), thus to accurately tune the slope used as boundary for the increase of response time and the decrease of throughput.

Since we cannot avoid thresholds in antipattern definitions, the detection accuracy is as good as the number and the accuracy of thresholds an antipattern requires. In this direction, we are working on defining a metrics that quantifies the degree of uncertainty of the approach. Such metrics is conceived as a function of the number and the type of thresholds a formalization requires. This metrics can be used to limit the detection of false positive/negative antipattern instances. Moreover, some fuzziness can be introduced for the evaluation of threshold values [39]. This might be useful to make

basic predicates more flexible, and to detect the performance flaws with diverse accuracy levels.

Finally, the process followed in the system element identification and classification brought us to specify performance antipatterns at the software architecture level independently from the notation used to model the (software) system and from the specific performance model and analysis techniques used to evaluate the performance indices. Hence, with appropriate techniques for exporting the system information, the approach has the potential to be applied to a wide range of software development and performance analysis methodologies.

8 Related work

The term *Antipattern* appeared for the first time in [9] in contrast to the trend of focus on positive and constructive solutions. Differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences.

Antipatterns have been applied in different domains. For example, in [42] data-flow antipatterns help to discover errors in workflows and are formalized through the CTL* temporal logic. As another example, in [8] antipatterns help to discover multi threading problems of java applications and are specified through the LTL temporal logic.

Performance Antipatterns, as the same name suggests, deal with performance issues of the software systems. They have been previously documented and discussed in different works: *technology-independent* performance antipatterns have been defined in [37] and they represent the main references in our work; *technology-specific* antipatterns have been specified in [16,41].

Enterprise technologies and EJB antipatterns are analyzed in [31]: antipatterns are represented as a set of rules loaded into the JESS [2] rule engine. The monitoring of the software application leads to reconstruct its run-time design and to obtain JESS facts. The matching between pre-defined rules and application facts is performed to carry out the detection of antipatterns.

Another recent work on the software performance diagnosis and improvements is proposed in [48]. Rules able to identify patterns of interaction between resources are defined, and again specified as JESS rules. Performance problems are identified before the implementation of the software system, even if they are based only on bottlenecks (e.g. the "One-Lane Bridge" antipattern) and long paths. Layered resource architectures are considered and the performance analysis is conducted with Layered Queueing Network (LQN) models. Such approach applies only to LQN models, hence its portability to other notations is yet to be proven and it may be quite complex. Our intent is instead to address a much wider set of modeling notations. This is the reason why we propose to represent performance antipatterns with basic constructs such as logical predicates.

Antipattern representation is a more recent research topic, whereas there has already been a significant effort in the area of representing software design *patterns*.

In [40], the authors propose a pattern specification language that is aimed to specify static and dynamic features of design patterns. Each pattern is analyzed from two complementary views: the structural and the behavioral views. In the context of performance antipatterns, we need an additional view, as the deployment view is necessary to represent platform properties.

In [27] design patterns are represented by logic predicates through which it is possible to identify roles and subsequently candidates for the patterns. Starting from the UML class diagram of the design pattern, it is possible to identify the role elements of the pattern and their relationship: each role is translated into a logical predicate, and finally the design pattern is a logical predicate that manages the interplay of all the different roles involved in its specification. Note that another benefit of using logical predicates is that they can be further refined on the basis of probabilistic model checking techniques, as Grunske experimented in [21].

In [18], a set of modeling patterns are used to explore the design space of soft real-time systems. Patterns are proposed as parametric templates that can be applied by setting appropriate values according to different deployments. A modeling language formalizes patterns representing design solutions that may have a different impact on the system performance intended as the amount of fulfilled real-time deadlines.

In [17], a metamodeling approach to pattern specification and detection has been introduced. In the context of the OMG's 4-layer metamodeling architecture, the authors propose a pattern language (i.e. Epattern, at the M3 level) to specify patterns in any MOF-compliant modeling language at the M2 layer. The Epattern language defines a pattern as a metaclass, which can then be inherited to create pattern hierarchies or composed to create larger patterns. A pattern is specified graphically using a UML composite structure-like diagram. No complete detection algorithm from the pattern specification is provided.

The specification of UML-based patterns is addressed in [19] where a pattern specification technique is aimed at defining design patterns as models in terms of UML metamodel concepts. A pattern model describes the participants of a pattern and the relations between them in a graphic notation by means of roles, i.e. the properties that a UML model element must have to match the corresponding pattern occurrence. In other words, a pattern is viewed as a meta-model (with structural and behavioral features) thus each instance of the pattern is a model in UML. Moreover, in [19] the pattern detection,

consisting on binding pattern models with system models, is not automated. Additionally, in the context of performance antipatterns, static and dynamic properties are not enough for representing performance issues. Performance annotations as well as the deployment reporting platform properties are necessary to target the performance evaluation of systems.

Search-based approaches [22] have been introduced to explore the problem space by examining options to deal with performance flaws. In [49], an approach to explore the design space and to find optimal deployment and scheduling priorities for tasks in a class of distributed real-time systems has been presented. However, its scope is restricted to improve the priorities of tasks competing for a processor, and the only refactoring action is aimed at changing the allocation of tasks to processors. In [29] meta-heuristic search techniques have been used for improving quality attributes of component-based software systems. This approach is quite time-consuming if the search is not guided by proper factors, such as the violation of performance requirements. A combination of meta-heuristic techniques and antipattern detection would be interesting to experiment to reduce the search space.

9 Conclusions

This work is a contribution to the backward path from performance analysis results to architectural feedback. Specifically, we have introduced an approach based on performance antipatterns that helps to identify the performance critical elements of software under development.

Performance antipatterns have been always considered as very useful examples of bad design practices, but little effort has been spent to formalize this knowledge and make it usable through automated instruments. The results of this paper show that this is a viable direction, thus more effort shall be dedicated to the antipattern manipulation to completely automate the backward path.

As the reader can realize, several open issues should be addressed in performance antipattern specification. However, the presented approach is the first thorough one at the best of our knowledge, so it represents a starting point for the implementation of the result interpretation and feedback generation step at the software architecture level.

The approach that we have introduced here for the antipattern representation is notation-independent, so it can be mapped onto various modeling languages through model transformation techniques. In fact, we are working in this direction to introduce a metamodeling approach to the antipattern definition and to use model-driven engineering techniques to detect and solve performance antipatterns.

Since an antipattern is made of a problem description and a solution description, besides the open issues pointed out in Sect. 7, we also are working on the antipattern solution. In particular, our work is inspired by existing languages for software refactoring based on patterns to define a refactoring representation for performance antipatterns.

A more complex problem to be faced for antipattern solution occurs whenever several antipatterns are detected in a model and heuristic approaches have to be applied to decide which antipatterns to solve among the found ones. Such problem has been partially tackled in [14], but many interesting issues have still to be faced, such as the simultaneous solution of multiple antipatterns.

This solution process can be quite complex, and the antipattern formalization is only the first step. For example, metrics can be introduced to quantify the role of views in the definition of an antipattern. In fact, once defined, an antipattern must be searched, often with incomplete information available to analysts. Therefore, the searching process is basically driven from heuristics, and metrics are crucial to effectively drive such process.

Finally, some critical pending issues have to be faced to automate the solution process. The solution of antipatterns in fact generates three main categories of problems: (i) the convergence problem, i.e. the solution of one or more antipatterns may introduce new antipatterns; (ii) the requirement problem, i.e. one or more antipatterns may not be solvable due to pre-existing (functional or non-functional) requirements; (iii) the coherency problem, i.e. the solution of a certain number of antipatterns may not be unambiguously applied due to incoherencies among their solutions.

Acknowledgments We would like to thank the anonymous reviewers for their useful comments that have helped us to improve the paper quality. This work has been partially supported by VISION ERC project (ERC-240555).

Appendix A: Identifying the foundational elements of antipatterns

In this appendix, we provide a structured description of the architectural model elements that occur in the definitions of antipatterns [37] (such as software entity, hardware utilization, operation throughput, etc.), which is meant to be the basis for a definition of antipatterns as logical predicates (see Sect. 4).

Since an (annotated) software architectural model contains details that are not relevant for the antipattern definition, as a first step we have filtered the concepts we need for the antipatterns representation. These concepts have been organized in an XML Schema, i.e. aimed at detecting performance issues, not representing software systems. The choice of XML [47] as representation language comes from its primary nature of interchange format supported by many tools that makes it one of the most straightforward means to define generic structured data.

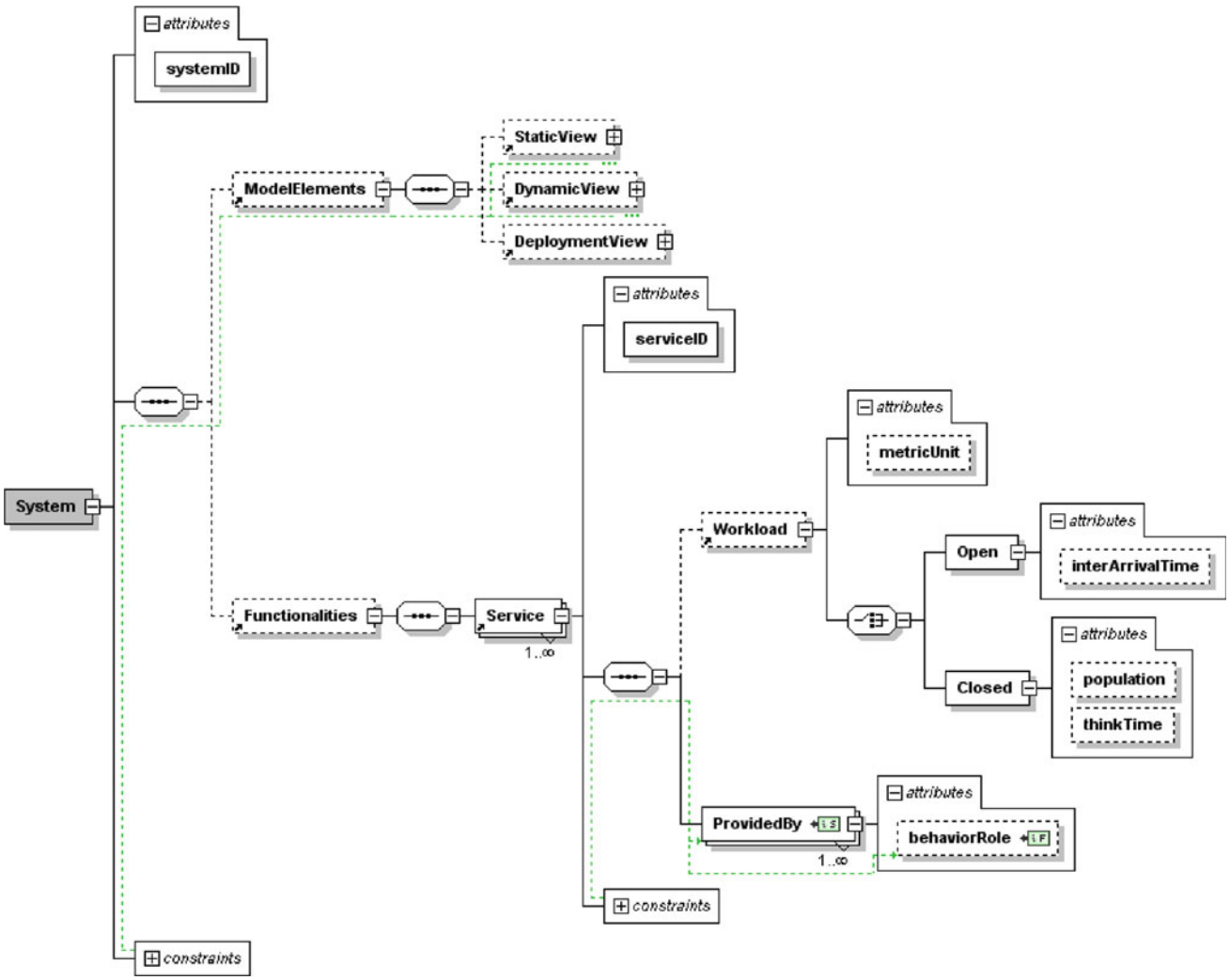


Fig. 19 An excerpt of the XML Schema

Software architectural model elements are organized in views. We consider three different views representing three sources of information: the *Static View* that captures the modules (e.g. classes, components, etc.) involved in the software system and the relationships among them; the *Dynamic View* that represents the interactions that occur between the modules to provide the system functionalities; and finally the *Deployment View* that describes the mapping of the modules onto platform sites. This organization stems from the Three-View Model that was introduced in [45] for performance engineering of software systems.

Overlaps among views can occur. For example, the elements interacting in the dynamics of a system are also part of the static and the deployment views. In particular, we adopt the term *Service* to represent the high-level functionalities of the software system that are meant to include all the interacting elements among the three views. To avoid redundancy and consistency problems, concepts shared by multiple views are defined once in a view, and simply referred in the other ones (through XML RFID).

The XML Schema we propose for performance antipatterns[8] is synthetically shown in Fig. 19: a *System* has an identifier (*systemID*) and it is composed of a set of *ModelElements* belonging to the three different Views, and of the set of *Functionalities* it provides. The *Static* View groups the elements needed to specify structural aspects of the software system; the *Dynamic* View deals with the behavior of the system; and finally the *Deployment* View captures the elements of the deployment configuration.

A *Service* has an identifier (*serviceID*) and it can be associated with a *Workload*, i.e. *Open* (specified by the *interArrivalTime*, e.g. a new request arrives each 0.5 s) or *Closed* (specified by the *population* and the *thinkTime*, e.g. there

[8] The XML Schema can be downloaded in http://www.di.univaq.it/catia.trubiani/phDthesis/PA_xmlSchema.xsd.

is a population of 25 requests entering the system, executing their operational profile, and then re-entering the system after a think time of 2 min). Note that the measurement unit (e.g. micro seconds, seconds, minutes) for the workload is specified in the *metricUnit* attribute. In fact, the *metricUnit* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *ms* (referring to micro seconds), *s* (referring to seconds), *min* (referring to minutes), and *other* (referring to any other metric unit customized by the user).

Each service contains static, dynamic and deployment elements; the reference to the architectural model elements is obtained by specifying that a service is *ProvidedBy* a set of *Behavior*s whose id reference (the behaviorID attribute, see Fig. 24a) is referred in the attribute *behaviorRole*. It will be the behavior (belonging to the Dynamic View) to contain all the other references among the model elements belonging to the other views (i.e. Static and Deployment).

A.1 Static view

The *Static View* contains elements to describe the static aspects of the system, it is composed of a set of *SoftwareEntity* and *Relationship* model elements.

The *SoftwareEntity* element has an identifier (*softwareEntityID*), a boolean value to specify if it is a database (*isDB*), an integer value to specify its pool size (*capacity*). A software entity contains a set of *SoftwareEntityInstance* elements specified by their identifiers (*softwareEntityInstanceID*), and a set of *Operation*(s).

A *Relationship* has an identifier (*relationshipID*), its *multiplicity*, and it contains a *Client* and a *Supplier*. Both these elements have an attribute, i.e. *clientRole* and *supplierRole*, respectively, that refers to a software entity instance identifier (*softwareEntityInstanceID*) previously declared. From a performance perspective, the only interesting relationship between two software entities is the usage relationship. Thus, the XML Schema does not contains elements for all relationships between two software entities (e.g. association, aggregation, etc.), but it flattens any of them in a client/supplier one. For example, in the aggregation relationship, a software entity aggregates one or more instances of another software entity to use their methods. In our Schema, this is represented as a *Relationship* where the first software entity instance has a *clientRole* and the aggregated ones has a *supplierRole*. In fact, we recall that the XML Schema we propose is not meant to represent software systems, but only to organize all the concepts providing useful information for the performance antipatterns.

The details of the *Operation* element are shown in Fig. 20b. An *Operation* has an identifier (*operationID*), and the *probability* of its occurrence. Besides, an operation has a *StructuredResourceDemand* composed of multiple *BasicResourceDemand*(s). A basic resource demand is composed of the resource *type* (e.g. cpu work units, database accesses, and network messages), and its *value* (e.g. number of cpu instructions, number of DB accesses, and number of messages sent over the network). Note that the resource *type* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *computation* (referring to cpu work units), *storage* (referring to database accesses), *bandwidth* (referring to network messages), and *other* (referring to any other resource type customized by the user).

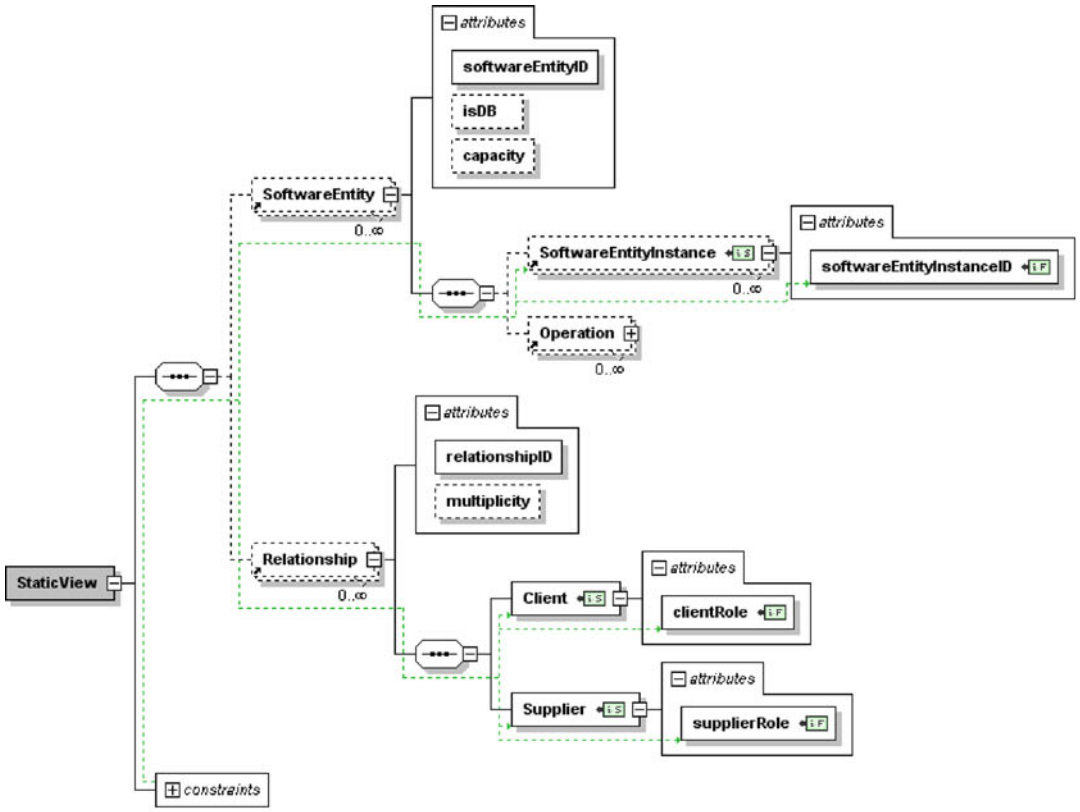
An operation contains a set of *OperationInstance*s specified with their identifiers (*operationInstanceID*), and some performance metrics are associated with each instance, namely *PerformanceMetrics*. The *metricUnit* attribute denotes the measurement unit adopted (e.g. micro seconds, seconds, minutes), and they can be either *Throughput* or *ResponseTime*. Multiple values can be specified for these two latter indices, and they represent the results from the simulation or the monitoring of the system over time. In fact, they can be evaluated at a certain date and/or time (*timestamp*), thus to capture the multiple-values antipatterns.

Figures 21, 22 and 23 show some examples of elements belonging to the Static View of the XML Schema: the left sides of these Figures give graphical representations of a software architectural model, whereas the right sides report excerpts of the XML files (compliant to the XML Schema) representing the features of the software architectural model.

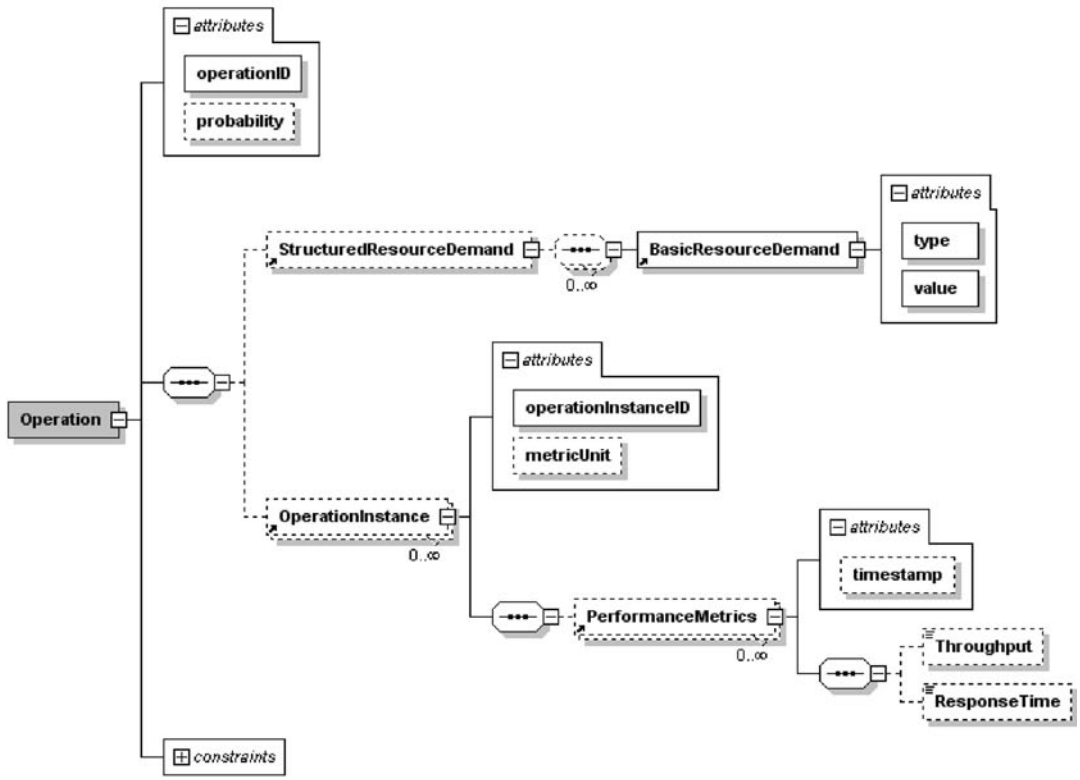
Figure 21 shows an example of the *Relationship* element. There are two software entity instances, i.e. the webServer $w1$ with a pool size of ten requests, and the database $d1$, and they are related each other through many connections. The *Relationship* involves the webServer in the *clientRole* and the database in the *supplierRole*; the number of operations required from the client to the supplier (i.e. getUserName, getUserAddress, getUserPassport, getUserWorkInfo), that is four, is stored in the *multiplicity* attribute of the *Relationship* element.

Figure 22 shows an example of the *StructuredResourceDemand* element. There is a Service (authorizeTransaction) that is provided by means of three operations (i.e. validateUser, validateTransaction, sendResult). The operation validateUser requires the following amount of resource demand: 1 *computation* unit, 2 *storage* units, and 0 *bandwidth* unit. We recall that computation, storage, and bandwidth represent the enumeration values for the basic resource demand *type* element of the XML Schema. The *Operation* validateUser that has two *OperationInstance*s (i.e. v1 and v2) that will be referred in Fig. 23.

Figure 23 shows an example of the *PerformanceMetrics* element. The *OperationInstance*s v1 and v2 are deployed on different processing nodes, i.e. proc1 and proc2. Note that performance metrics can be obtained by simulating or monitoring the performance model or by solving it analytic or other ways. In the example that we propose in



(a) *SoftwareEntity* and *Relationship* elements.



(b) *Operation* element.

Fig. 20 XML schema—*Static View*

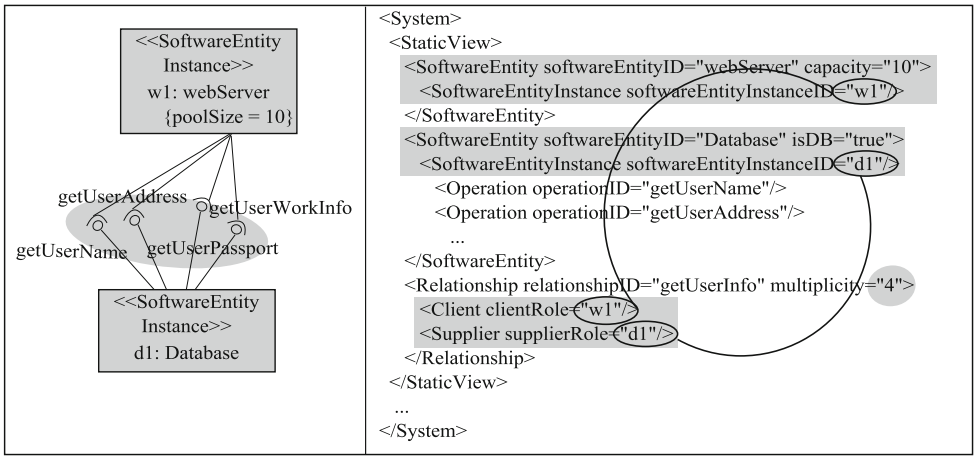


Fig. 21 An example of the *Relationship* element

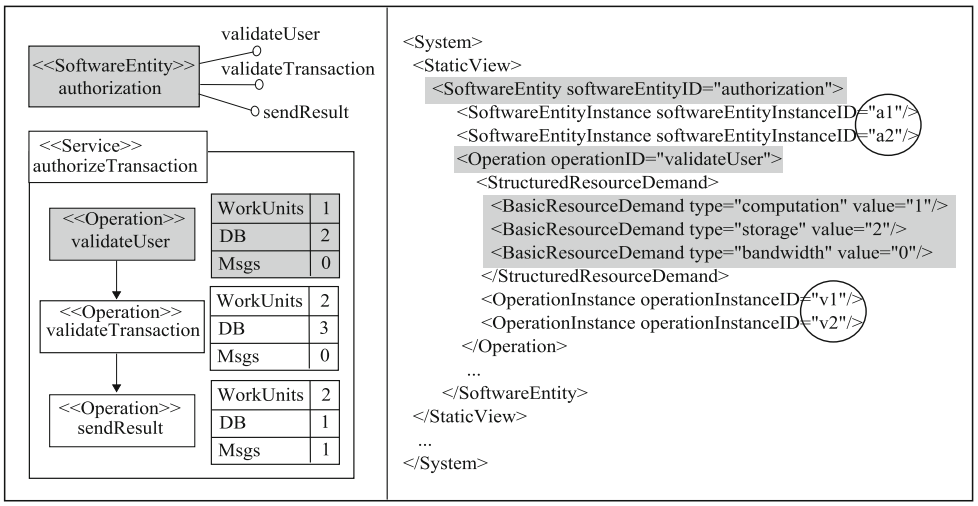


Fig. 22 An example of the *StructuredResourceDemand* element

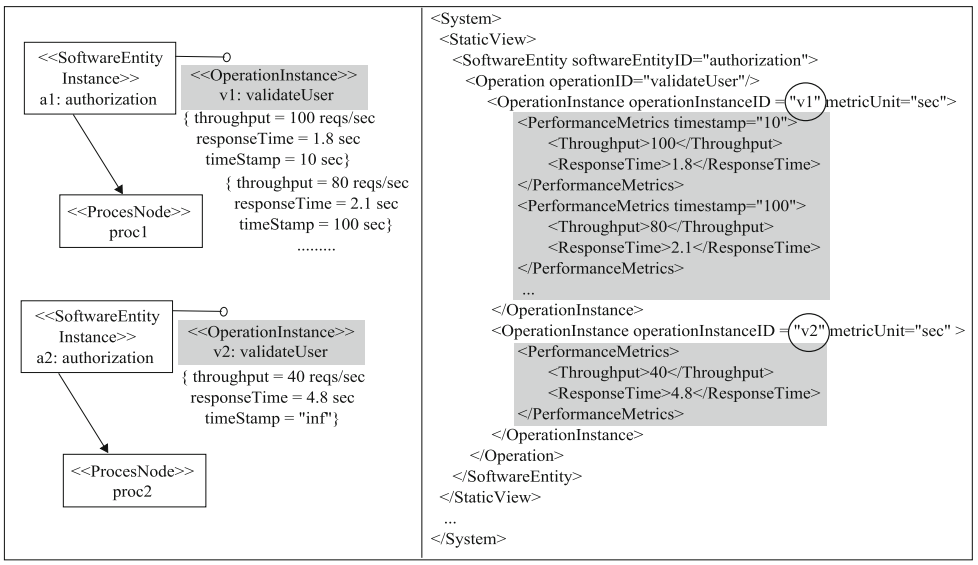


Fig. 23 An example of the *PerformanceMetrics* element

Fig. 23, both solutions are applied: (i) the *OperationInstance* v1 is evaluated by simulating the performance model, and the simulation reveals that after the system is running for 10 s there are the following performance metrics: *Throughput* = 100requests/s, *ResponseTime* = 1.8 s; whereas after the system is running for 100 s there are the following performance metrics: *Throughput* = 80 requests/s, *ResponseTime* = 2.1 s; (ii) the *OperationInstance* v2 is evaluated by

solving the performance model in an analytic way, and the analytic solution reveals that there are the following performance metrics: *Throughput* = 40 requests/s, *ResponseTime* = 4.8 s. We recall that ms, s, and min represent the enumeration values for the operation instance *metricUnit* element of the XML Schema.

A.2 Dynamic view

The *Dynamic View* (Fig. 24a) is made of *Behavior*(s). Each behavior has an identifier (*behaviorID*), and its execution *probability*. A *Behavior* contains either a set of *Message*(s) or an *Operator*.

As shown in Fig. 24a, an *Operator* can be either a behavior *Alternative* with the *probability* it occurs or a *Loop* with the number of iterations (*no_of_iterations*). An operator might contain *Message*(s) and optionally another nested *Operator*.

As shown in Fig. 24b, a *Message* is described by the *multiplicity* of the communication pattern, the *size* and its *sizeUnit* (e.g. Kilobyte, Megabyte, Gigabyte), and the *format* (e.g. xml) used in the communication. Note that the *sizeUnit* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *Kb* (referring to Kilobyte), *Mb* (referring to Megabyte), *Gb* (referring to Gigabyte), and *other* (referring to any other size unit customized by the user). From a performance perspective, the size of messages is useful to detect antipatterns in message-based systems that require heavy communication overhead for sending messages with a very small amount of information, whereas the format of messages is useful when the sender of the message translates it into an intermediate format, and then the receiver parses and translates it in an internal format before processing it. The translation and parsing of formats could be time consuming, thus degrading the performance.

A message has a *Sender* identified by *senderRole*, a *Receiver* identified by *receiverRole*, and they both refer to a *softwareEntityInstanceID* playing such role. Additionally, it is possible to specify if the message is *Synchronous* or *Other*. We are not interested to asynchronous and reply messages, we flatten them in *Other*. Also in this case we can notice that the XML Schema we propose it is not meant to represent software systems, but to keep all the concepts providing useful information for the antipatterns. In fact, synchronous messages are of particular interest for the One-Lane Bridge antipattern, since it occurs when a set of processes make a synchronous call to another process that is not multithreaded, hence only a few processes may continue to execute concurrently. Note that *Other* is intentionally added to enrich the vocabulary of the XML Schema, and it can be further detailed if new antipatterns are considered.

The *Task* determines the invocation of one operation, it is characterized by an identifier (*taskRole*) that refers to an *operationInstanceID* playing such role. Optionally there may be an attribute to specify the semantic of the message (i.e. *IsCreateObjectAction* or *IsDestroyObjectAction*) that allows the detection of frequent and unnecessary creations and destructions of objects belonging to the same software entity instance.

Figure 25 shows an example of elements belonging to the Dynamic View of the XML Schema: the left side of the Figure gives a graphical representation of a software architectural model, whereas the right side reports an excerpt of the XML file (compliant to the XML Schema) representing the features of the software architectural model. In particular, Fig. 25 shows an example of the *Behavior* element. There are two software entity instances (a1 and a2) exchanging the user credentials, and the behavior is performed as follows. The software entity instance a1 sends a synchronous message (with a size of 0.5 kilobyte and xml format) to the software entity instance a2 by invoking the execution of the operation validateUser. Note that the *taskRole* attribute of Fig. 25 refers to the *operationInstanceID* of Fig. 23.

A.3 Deployment view

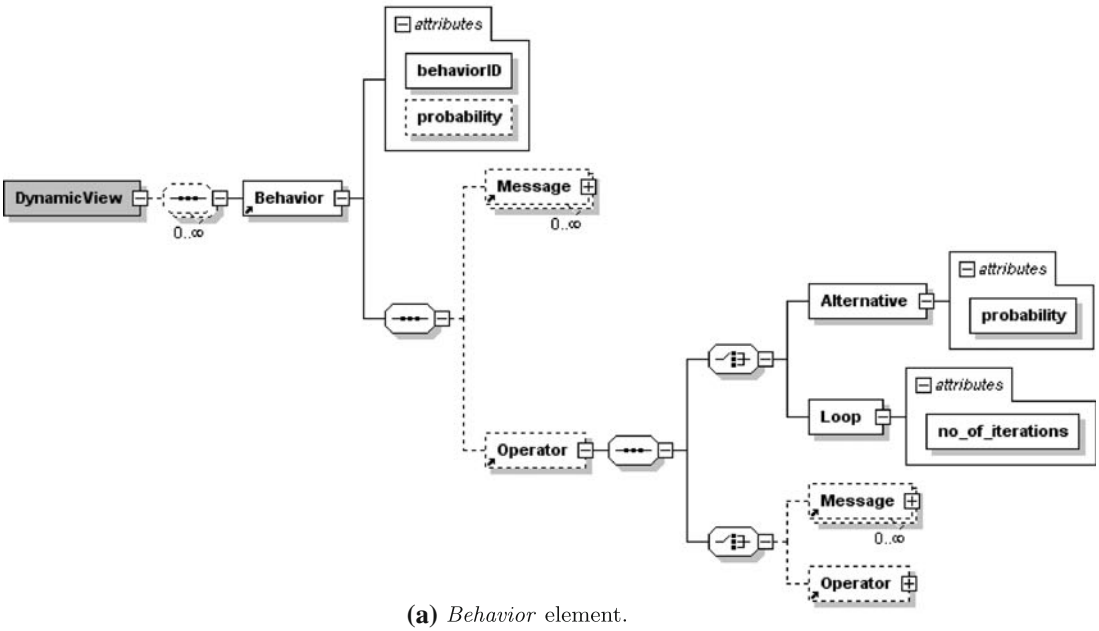
The *DeploymentView* (see Fig. 26a) is made of a set of processing nodes (*ProcesNode*), and optional *NetworkLink*(s) that enable the communication between the nodes.

As shown in Fig. 26b, a processing node has an identifier (*procesNodeID*), and it contains a set of *DeployedInstance*(s). Each *DeployedInstance* has an identifier (*deployedInstanceRole*) that refers to a *softwareEntityInstanceID* already defined in the Static View.

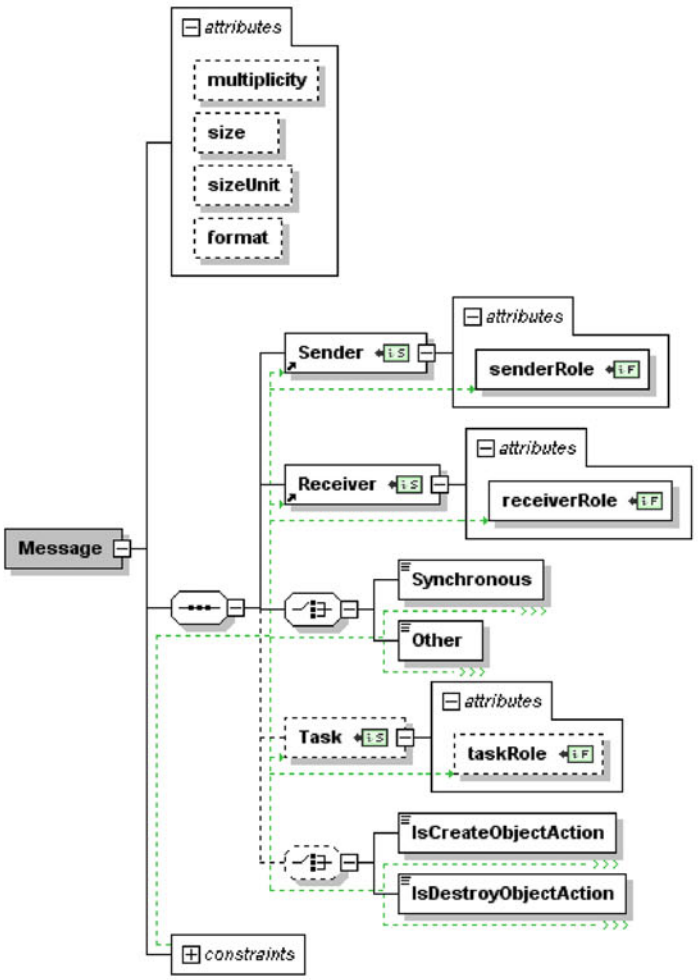
A processing node additionally contains a set of hardware entities (*HardwareEntity*). Each hardware entity has an identifier (*hardwareEntityID*), the *Utilization* and the *QueueLength* values, and the *type* indicating whether it is a *cpu* or a *disk*. In fact, the *type* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *cpu* (referring to cpu devices), *disk* (referring to disk devices), and *other* (referring to any other type customized by the user). From a performance perspective, the distinction between cpu(s) and disk(s) is useful to point out the type of work assigned to a processing node by checking their utilization values (e.g. a database transaction uses more disk than cpu).

Specific performance metrics for a processing node are also defined, i.e. *ProcNodePerfMetrics*, such as its *ServiceTime* (e.g. the average processing time used to execute requests incoming to the processing node is 2 s), and its *WaitingTime* (e.g. the average waiting time for requests incoming to the processing node is 5 s). The *metricUnit* attribute denotes again the measurement unit adopted (e.g. micro seconds, seconds, minutes).

From the performance perspective, some additional information can be useful in this view. It may happens that while



(a) *Behavior* element.



(b) *Message* element.

Fig. 24 XML schema—*Dynamic View*

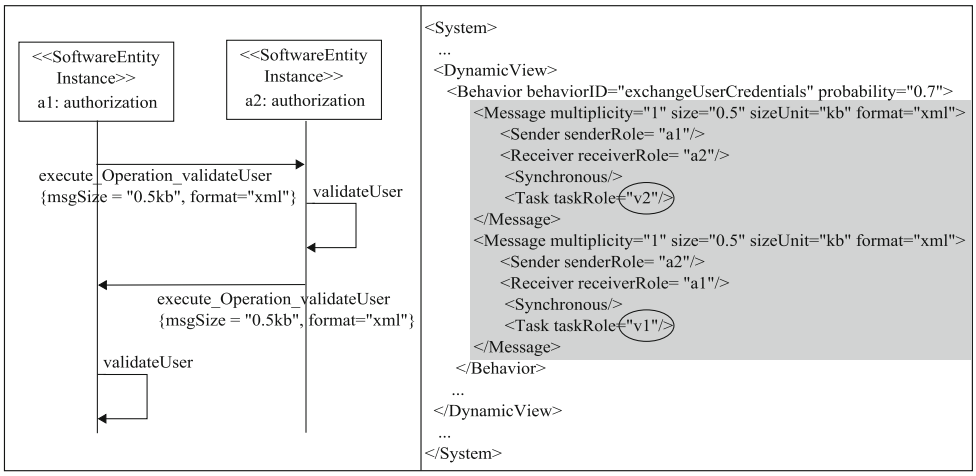


Fig. 25 An example of the *Behavior* element

trying to run too many programs at the same time, this introduces an extremely high paging rate, thus systems spend all their time serving page faults rather than processing requests. To represent such scenario, we introduce a set of parameters, *Params*, that the processing nodes can manage: the number of database connections (*dbConnections*), the number of internet connections (*webConnections*), the amount of pooled resources (*pooledResources*), and finally the number of concurrent streams (*concurrentStreams*). All these parameters are associated with a certain *timestamp* to monitor their trend along the time.

A *NetworkLink* (see Fig. 26a) has an identifier (*networkLinkID*), and two or more *EndNode*s. Each end node has an identifier (*endNodeRole*) that refers to *procesNodeID* playing such role. It optionally contains information about the network: the available bandwidth (*capacity*) denoting the maximum message size it supports, its utilization (*usedBandwidth*), and the *bitRate* used in the communication. Note that the *bitRate* element uses an enumeration aimed at specifying a set of valid values for that element, i.e. *Kbit/s* (referring to Kilobit/second), *Mbit/s* (referring to Megabit/second), *Gbit/s* (referring to Gigabit/second), and *other* (referring to any other bit rate unit customized by the user). From a performance perspective, the bit rate of the network is useful to detect antipatterns in message-based systems that require heavy communication overhead for sending messages with a very small amount of information.

Figures 27 and 28 show some examples of elements belonging to the Deployment View of the XML Schema: the left sides of the Figures give graphical representations of a software architectural model, whereas the right sides report an excerpt of the XML files (compliant to the XML Schema) representing the features of the software architectural model.

Figure 27 shows an example of the *NetworkLink* element. There are four processing nodes, i.e. proc1, …, proc4, communicating through two network links: the *NetworkLink* net1 allows the communication among the processing nodes proc1, proc2, and proc3; the *NetworkLink* net2 allows the communication among the processing nodes proc3, and proc4. For each network link it is specified the bandwidth and the utilization; for example, net1 has a *capacity* of 100 with a *bitRate* expressed in Megabit/second and its *usedBandwidth* is equal to 0.42.

Figure 28 shows an example of the *ProcesNode* element. There is a processing node (proc1) with three hardware entities, i.e. two cpus (proc1_cpu1, proc1_cpu2) and one disk (proc1_disk1). For each hardware entity, it is possible to specify the queue length (qL) and the utilization; for example, the proc1_cpu1 hardware entity reveals an average utilization of 0.3, and an average queue length of 30 users. Additionally, some performance metrics are evaluated for the processing node: the *serviceTime* keeps the time necessary to perform a task (i.e. 1.5 s in proc1), and the *waitingTime* stores how long incoming requests to the node must wait before being processed (i.e. 0.01 s in proc1).

In this appendix, we provided a generic data structure (an XML Schema) collecting all the architectural model elements that occur in the definitions of antipatterns [37] (such as software entity, hardware utilization, operation throughput, etc.). It represents a groundwork for the definition of antipatterns as logical predicates (see Sect. 4), thus to achieve a formalization of the knowledge commonly encountered by performance engineers in practice.

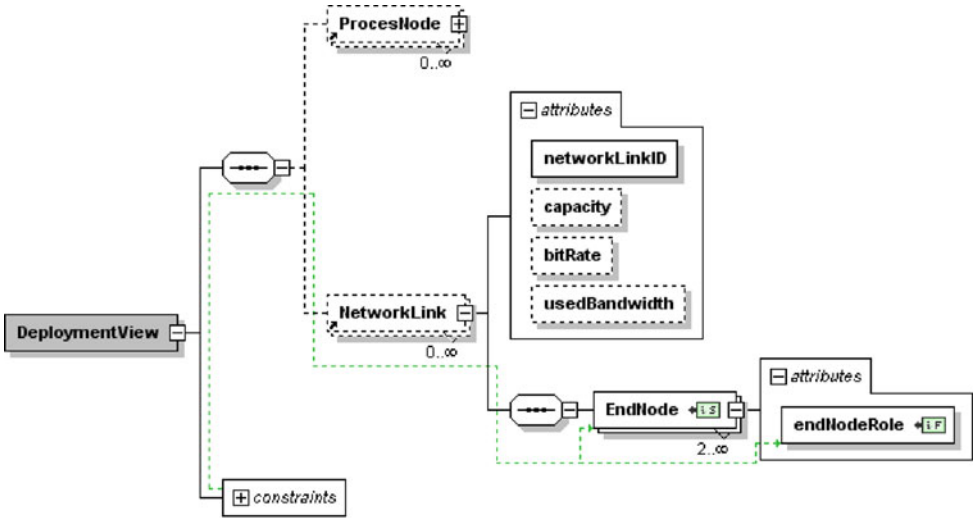
Appendix B: Performance antipatterns as logical predicates

In this appendix, we report the representation of the performance antipatterns not presented in Sect. 4.

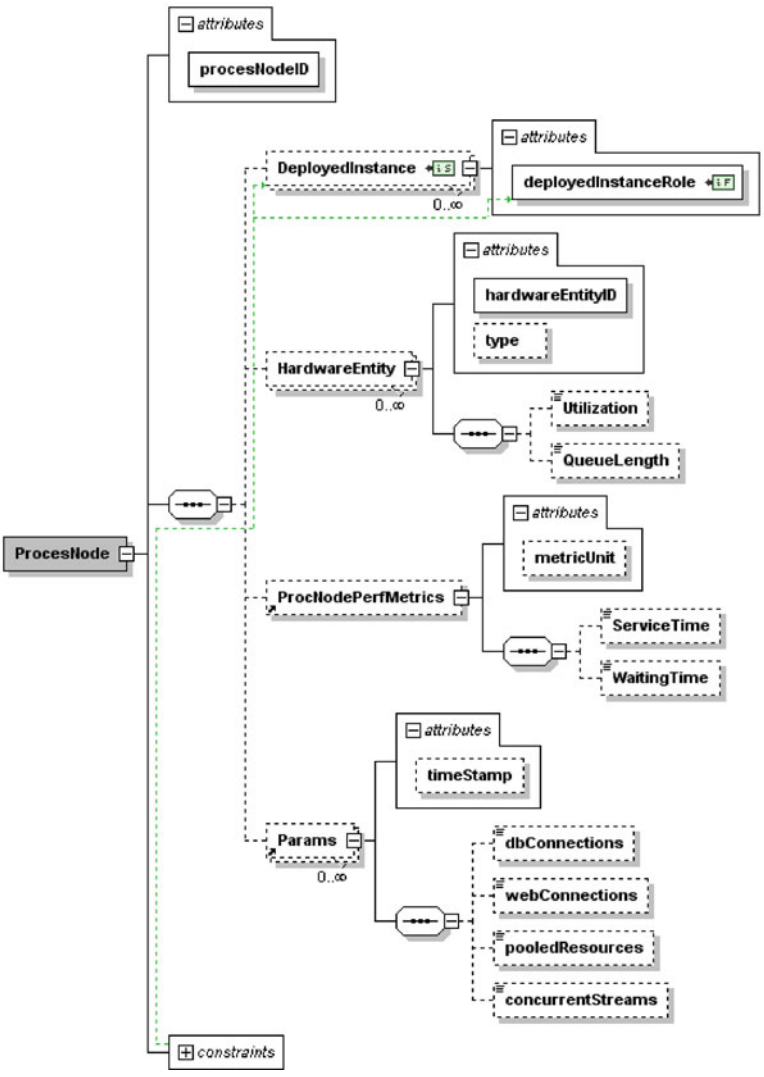
B.1 Unbalanced Processing

Pipe and Filter Architectures and Extensive Processing antipatterns are both manifestations of the unbalanced processing: "*Imagine waiting in an airline check-in line. Multiple agents*

Fig. 26 XML schema—*Deployment View*



(a) *NetworkLink* element.



(b) *ProcesNode* element.

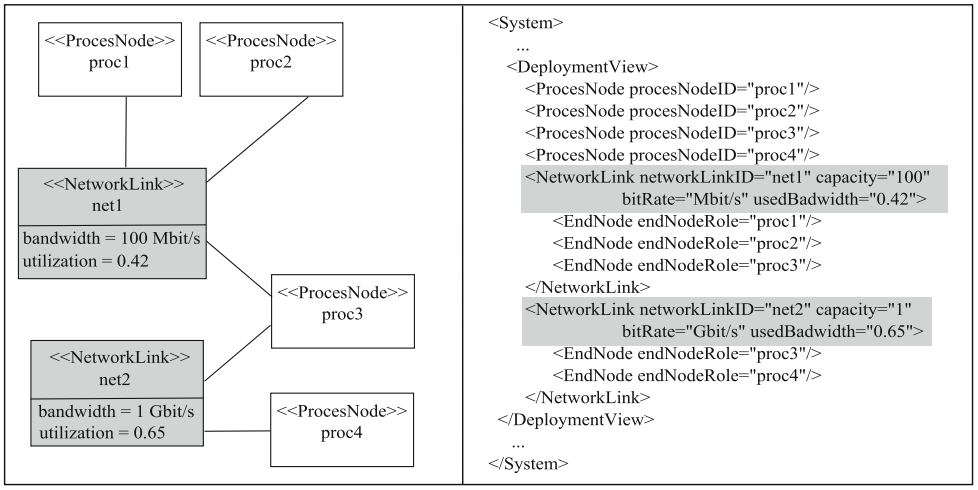


Fig. 27 An example of the *NetworkLink* element

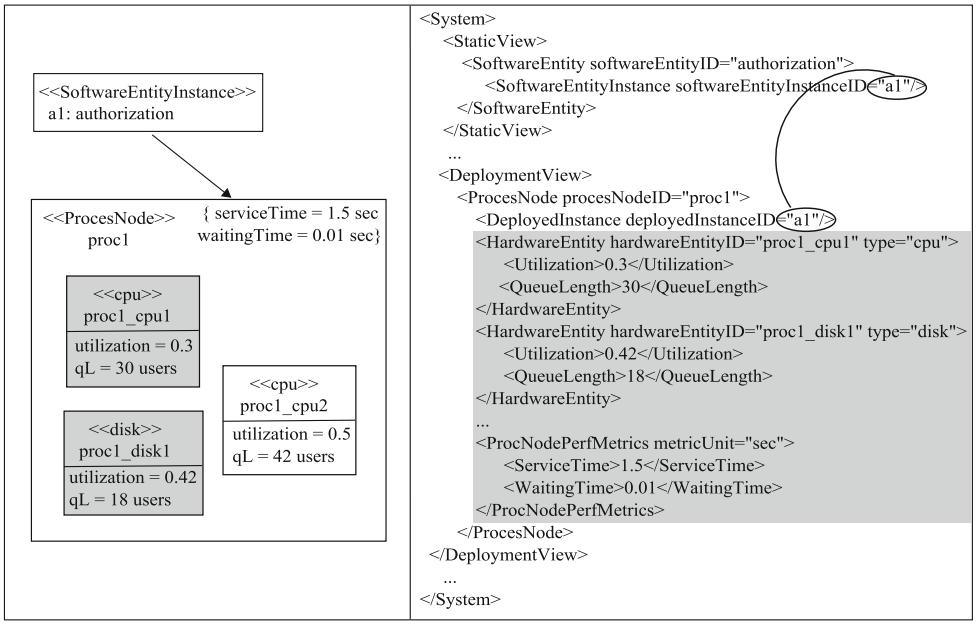


Fig. 28 An example of the *ProcesNode* element

can speed-up the process but, if a customer needs to change an entire itinerary, the agent serving him or her is tied-up for a long time making those changes. With this agent (processor) effectively removed from the pool for the time required to service this request, the entire line moves more slowly and, as more customers arrive, the line becomes longer" quoted by [35].

B.2 "Pipe and Filter" Architectures

"Pipe and Filter" Architectures [35] has the following problem informal definition: *"occurs when the slowest filter in a pipe and filter architecture causes the system to have unacceptable throughput"* (see Table 1).

"*For example, in the travel analogy, passengers must go through several stages (or filters): first check in at the ticket counter, then pass through security, then go through the boarding process. Recent events have caused each stage to go more slowly. The security stage tends to be the slowest filter these days*" quoted by [35].

We formalize this sentence with four basic predicates: the BP_1, BP_2, BP_3 predicates whose elements belong to the Static View; the BP_4 predicate whose elements belong to the Deployment View.

BP_1 There is at least one `Operation` Op that represents the slowest filter, i.e. it requires a set of resource demands higher than a given thresholds set. Let us define by $F_{resDemand}$ the function providing the `StructuredResourceDemand` of the operation Op that returns an array of values corresponding to the resource demand(s) the operation Op requires:

$$\forall i : F_{resDemand}(Op)[i] \geq Th_{resDemand}[i] \quad (19)$$

BP_2 There is at least one `Service` S that invokes the `OperationInstance` OpI that is instance of Op. Let us define by $F_{probExec}$ the function that provides the probability of execution of the operation instance OpI when the service S is invoked. If it is equal to 1 it means that the `Task` referring to such operation is mandatory:

$$F_{probExec}(S, OpI) = 1 \tag{20}$$

BP_3 The throughput of the service S is unacceptable, i.e. lower than the value defined by an user requirement Th_{SthReq}. Let us define by F_T the function that returns the `Throughput` of the service S:

$$F_T(S) < Th_{SthReq} \tag{21}$$

BP_4 The `ProcesNode` P_{swE_x} on which the software instance swE_x (i.e. the software entity instance that offers OpI) is deployed has a heavy computation. For the formalization of this characteristic, we recall the $F_{maxHwUtil}$ function with the 'all' option that returns the maximum `Utilization` among all the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \tag{22}$$

Summarizing, the *"Pipe and Filter" Architectures* antipattern occurs when the following composed predicate is true:

$$\exists OpI \in \mathbb{O}, S \in \mathbb{S} \mid \boxed{(19) \wedge (20) \wedge ((21) \vee (22))}$$

where $\mathbb{O}$ represents the set of all the `OperationInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (OpI, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a "Pipe and Filter" Architectures antipattern.

B.3 Extensive Processing

Extensive Processing [35] has the following problem informal definition: *"occurs when extensive processing in general impedes overall response time"* (see Table 1).

"*This situation is analogous to the itinerary-change example. It occurs when a long running process monopolizes a processor. The processor is removed from the pool, but unlike the pipe and filter example, other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload*" quoted by [35].

We formalize this sentence with four basic predicates: the BP_1, BP_2, BP_3 predicates whose elements belong to the Static View; the BP_4 predicate whose elements belong to the Deployment View.

BP_1 There are at least two `Operations` Op_1 and Op_2 such that: (i) Op_1 has a resource demand vector higher than an upper bound threshold vector (23a); (ii) Op_2 has a resource demand vector lower than a lower bound threshold vector (23b). The `StructuredResourceDemand` of the operations is provided by the function $F_{resDemand}$ that returns an array of values corresponding to the resource demand(s) the operations Op_1 and Op_2 require. In practice:

$$\forall i : F_{resDemand}(Op_1)[i] \geq Th_{maxOpResDemand}[i] \tag{23a}$$

$$\forall i : F_{resDemand}(Op_2)[i] < Th_{minOpResDemand}[i] \tag{23b}$$

BP_2 There is at least one `Service` S that invokes the `OperationInstances` OpI_1 (instance of Op_1), and OpI_2 (instance of Op_2). Unlike to the "Pipe and Filter" Architectures, OpI_1 and OpI_2 are alternately executed in the service S. This condition can be formalized using the $F_{probExec}$ function that returns the probability of execution of the operation instances OpI_1 and OpI_2 when the service S has been invoked, such that:

$$F_{probExec}(S, OpI_1) + F_{probExec}(S, OpI_2) = 1 \tag{24}$$

BP_3 The response time of the service S is unacceptable, i.e. larger than the value Th_{SrtReq} defined by a user requirement. Let us define by F_{RT} the function that returns the `ResponseTime` of the service S:

$$F_{RT}(S) > Th_{SrtReq} \tag{25}$$

BP_4 The `ProcesNode` P_{swE_x} on which the software instance swE_x (i.e. the software entity instance that offers the operation instance OpI_1) is deployed has a heavy computation. For the formalization of this characteristic, we recall the $F_{maxHwUtil}$ function with the 'all' option that returns the maximum `Utilization` among all the hardware entities of the processing node. We compare such value with a threshold $Th_{maxHwUtil}$:

$$F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \tag{26}$$

Summarizing, the *Extensive Processing* antipattern occurs when the following composed predicate is true:

$$\exists OpI_1, OpI_2 \in \mathbb{O}, S \in \mathbb{S} \mid \boxed{(23a) \wedge (23b) \wedge (24) \wedge ((25) \vee (26))}$$

where $\mathbb{O}$ represents the set of all the `OperationInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (OpI_1, OpI_2, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents an Extensive Processing antipattern.

B.4 Tower of Babel

Tower of Babel [37] has the following problem informal definition: *"occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML"* (see Table 1).

We formalize this sentence with two basic predicates: the BP_1 predicate whose elements belong to the Dynamic View; the BP_2 predicate whose elements belong to the Deployment View.

BP_1 There is at least one `Service` S in which the information is often translated from an internal format to an exchange format, and back. Let us define by F_{numExF} the function that counts how many times the `format` is changed in the service S by a `SoftwareEntityInstance` swE_x. The antipattern can occur when this function returns a value higher or equal than a threshold Th_{maxExF}:

$$F_{numExF}(swE_x, S) \geq Th_{maxExF} \tag{27}$$

BP_2 The `ProcesNode` P_{swE_x} on which the software entity instance swE_x is deployed has a heavy computation. That is, the `Utilization` of hardware entities belonging to the `ProcesNode` P_{swE_x} exceeds a threshold value. For the formalization of this characteristic, we recall the $F_{maxHwUtil}$ function, with the 'all' option that returns the maximum `Utilization` among the ones of the hardware entities of the processing node:

$$F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil} \tag{28}$$

Summarizing, the *Tower of Babel* antipattern occurs when the following composed predicate is true:

$$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(27) \wedge (28)}$$

where $\text{sw}\mathbb{E}$ represents the set of all `SoftwareEntityInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (swE_x, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a Tower of Babel antipattern.

B.5 One-Lane Bridge

One-Lane Bridge [33] has the following problem informal definition: *"occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g. when accessing a database). Other processes are delayed while they wait for their turn"* (see Table 1).

We formalize this sentence with four basic predicates: the BP_1 and BP_2 predicates whose elements belong to the Dynamic View; the BP_3 predicate whose elements belong to the Static View.

BP_1 There is at least one `SoftwareEntityInstance` swE_x that receives a large number of synchronous calls, i.e. its `capacity` (i.e. the parallelism degree) is lower than the incoming requests rate in a service S. Let us define by $F_{numSynchCalls}$ the function providing the number of synchronous calls that swE_x receives for a service S, and by $F_{poolSize}$ the function providing the pool size capacity of swE_x:

$$F_{numSynchCalls}(swE_x, S) \gg F_{poolSize}(swE_x) \tag{29}$$

BP_2 The requests incoming to the processing node on which swE_x is deployed are delayed, the `ServiceTime` is much lower than the `WaitingTime`. Let us define by $F_{serviceTime}$ and $F_{waitingTime}$ the functions providing the service time and the waiting time, respectively, for the processing node P_{swE_x}:

$$F_{serviceTime}(P_{swE_x}) \ll F_{waitingTime}(P_{swE_x}) \tag{30}$$

BP_3 The response time of the service S is unacceptable, i.e. larger than the value Th_{SrtReq} defined by a user requirement. Let us define by F_{RT} the function that returns the `ResponseTime` of the service S:

$$F_{RT}(S) > Th_{SrtReq} \tag{31}$$

Summarizing, the *One-Lane Bridge* antipattern occurs when the following composed predicate is true:

$$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(29) \wedge (30) \wedge (31)}$$

where $\text{sw}\mathbb{E}$ represents the `SoftwareEntityInstances`, and $\mathbb{S}$ represents the `Services` in the software system. Each (swE_x, S) instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a One-Lane Bridge antipattern.

B.6 Excessive Dynamic Allocation

Excessive Dynamic Allocation [33] has the following problem informal definition: *"occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance"* (see Table 1).

We formalize this sentence with two basic predicates: the BP_1 predicate whose elements belong to the Dynamic View, and the BP_2 predicate whose elements belong to the Static View.

BP_1 There is at least one `Service` S in which objects are created in a "just-in-time" approach, when their capabilities are needed, and then destroyed when they are no longer required. Let us define by $F_{numCreatedObj}$ the function that calculates the number of created objects (i.e. `IsCreateObjectAction`), and $F_{numDestroyedObj}$ the function that returns the number of destroyed ones (i.e. `IsDestroyObjectAction`):

$$F_{numCreatedObj}(S) \geq Th_{maxCrObj} \tag{32}$$

$$F_{numDestroyedObj}(S) \geq Th_{maxDeObj} \tag{33}$$

BP_2 The overhead for creating and destroying a single object may be small, but when a large number of objects are frequently created and then destroyed, the response time may be significantly increased. Let us recall the function F_{RT} that

returns the ResponseTime of the service S. If such value is larger than the user requirement, then the antipattern can occur:

$$F_{RT}(S) > Th_{SrtReq} \tag{34}$$

Summarizing, the *Excessive Dynamic Allocation* antipattern occurs when the following composed predicate is true:

$$\exists S \in \mathbb{S} \mid \boxed{((32) \vee (33)) \wedge (34)}$$

where $\mathbb{S}$ represents the set of all the Services in the software system. Each S instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents an Excessive Dynamic Allocation antipattern.

B.7 The Ramp

The Ramp [37] has the following problem informal definition: *"occurs when processing time increases as the system is used"* (see Table 1).

We formalize this sentence with two basic predicates: the BP_1 and BP_2 predicates whose elements belong to the Static View.

BP_1 There is at least one OperationInstance OpI that has an increasing value for the response time along different observation time slots. Let us define by F_{RT} the function that returns the mean ResponseTime of the operation instance OpI observed in the time slot t. The Ramp can occur when the average response time of the operation instance increases in n consecutive time slots, which means that it is higher than a threshold $Th_{OpRtVar}$:

$$\frac{\sum_{1 \le t \le n} |F_{RT}(OpI, t) - F_{RT}(OpI, t-1)|}{n} > Th_{OpRtVar} \tag{35}$$

BP_2 The OperationInstance OpI shows a decreasing value for the throughput along different observation time slots. Let us define by F_T the function that returns the mean Throughput of the operation instance OpI observed in the time slot t. The Ramp occurs when the absolute value of the average throughput of the operation instance increases in n consecutive time slots, which means that it is higher than a threshold $Th_{OpThVar}$:

$$\frac{\sum_{1 \le t \le n} |F_{T}(OpI, t) - F_{T}(OpI, t-1)|}{n} > Th_{OpThVar} \tag{36}$$

Summarizing, *The Ramp* antipattern occurs when the following composed predicate is true:

$$\exists OpI \in \mathbb{O} \mid \boxed{(35) \wedge (36)}$$

where $\mathbb{O}$ represents the set of all the OperationInstances in the software system. Each OpI instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents The Ramp antipattern.

B.8 More is Less

More is Less [35] has the following problem informal definition: *"occurs when a system spends more time "thrashing" than accomplishing real work because there are too many processes relative to available resources"* (see Table 1).

We formalize this sentence with one basic predicate: the BP_1 predicate whose elements belong to the Deployment View.

BP_1 There is at least one ProcesNode P_x whose configuration parameters are not able to support the workload required to the software system. The parameters we refer are the number of concurrent *dbConnections*, the *webConnections*, the *pooledResources*, or the *concurrentStreams*.

Let us define by $F_{par}[i]$ the function that returns the ith configuration parameter defined for the system; and by $F_{RTpar}[i]$ the function returning the ith run time parameter observed in the time slot t. The More is Less antipattern can occur when the configuration parameters are much lower than the average values of the run time parameters in n consecutive time slots:

$$\forall i : F_{par}(P_x)[i] \ll \frac{\sum_{1 \le t \le n} (F_{RTpar}(P_x, t)[i] - F_{RTpar}(P_x, t-1)[i])}{n} \tag{37}$$

Summarizing, the *More is Less* antipattern occurs when the following predicate is true:

$$\exists P_x \in \mathbb{P} \mid \boxed{(37)}$$

where $\mathbb{P}$ represents the set of all the ProcesNodes in the software system. Each P_x instance satisfying the predicate must be pointed out to the designer for a deeper analysis, because it represents a More is Less antipattern.

Appendix C: Summary

This section summarizes the auxiliary *functions* and the *thresholds* introduced for representing antipatterns as logical predicates.

Table 10 reports the supporting functions we need to quantify specific aspects of the software architectural model elements. In particular, the first column of the Table shows the *signature* of the function and the second column provides its *description*. For example, in the first row the $F_{numClientConnects}$ function takes as input one software entity instance and returns an integer that represents the multiplicity of the relationships where swE_x assumes the client role.

Table 10 Functions specification

Signature	Description
int $F_{numClientConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships among software entity instances where swE_x is involved as client
int $F_{numSupplierConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships among software entity instances where swE_x is involved as supplier
int $F_{numMsgs}$ (SoftwareEntityInstance swE_x, SoftwareEntityInstance swE_y, Service S)	It counts the number of messages sent from swE_x to swE_y in a service S
float $F_{maxHwUtil}$ (ProcesNode pn_x, type T)	It provides the maximum hardware utilization among the hardware devices of a certain type T={cpu, disk, all} composing the processing node pn_x
float $F_{maxNetUtil}$ (ProcesNode pn_x, ProcesNode pn_y)	It provides the maximum utilization among the network links joining the processing nodes pn_x and pn_y
float $F_{maxNetUtil}$ (ProcesNode pn_x, SoftwareEntityInstance swE_x)	It provides the maximum utilization among the network links used by swE_x, deployed on pn_x, to interact with other software entity instances
float F_{maxQL} (ProcesNode pn_x)	It provides the maximum queue length among the hardware devices composing the processing node pn_x
int[ ] $F_{resDemand}$ (Operation Op)	It provides the resource demand of the operation Op
float $F_{probExec}$ (Service S, Operation Op)	It provides the probability the operation Op is executed in the service S
float F_T (Service S)	It provides the estimated throughput of the service S at the steady-state
float F_{RT} (Service S)	It provides the estimated response time of the service S at the steady-state
float F_T (Service S, timeInterval t)	It provides the estimated throughput of the service S at the time interval t
float F_{RT} (Service S, timeInterval t)	It provides the estimated response time of the service S at the time interval t
int $F_{numDBmsgs}$ (SoftwareEntityInstance swE_x, SoftwareEntityInstance swE_y, Service S)	It counts the number of requests generated by swE_x to the database swE_y in a service S
int $F_{numRemMsgs}$ (SoftwareEntityInstance swE_x, Service S)	It counts the number of remote messages sent by swE_x in a service S
int $F_{numRemInst}$ (SoftwareEntityInstance swE_x, Service S)	It provides the number of remote instances with which swE_x communicates in a service S
int F_{numExF} (SoftwareEntityInstance swE_x, Service S)	It provides the number of exchange formats performed by swE_x in a service S
int $F_{numSynchCalls}$ (SoftwareEntityInstance swE_x, Service S)	It provides the number of synchronous calls swE_x receives in a service S
int $F_{poolSize}$ (SoftwareEntityInstance swE_x)	It provides the pool size capacity of swE_x
float $F_{serviceTime}$ (ProcesNode pn_x)	It provides the service time of pn_x
float $F_{waitingTime}$ (ProcesNode pn_x)	It provides the waiting time of pn_x
int $F_{numCreatedObj}$ (Service S)	It provides the number of objects dynamically created in a service S
int $F_{numDestroyedObj}$ (Service S)	It provides the number of objects dynamically destroyed in a service S
int[ ] F_{par} (ProcesNode pn_x)	It provides the array of configuration parameters related to the pn_x processing node
int[ ] F_{RTpar} (ProcesNode pn_x, timeInterval t)	It provides the array of configuration parameters related to the pn_x processing node at the time interval t

Table 11 reports the thresholds we need to evaluate *software* boundaries. In particular, the first column of the table shows the name of the *threshold*, the second column provides its *description*, and finally in the third column it is proposed an *heuristics* to estimate its numerical value. For example, the $Th_{maxConnects}$ threshold represents the maximum bound for the number of usage relationships a software entity is involved in. It can be estimated as the average number of usage relationships, with reference to the entire set of software instances in the software system, plus the corresponding variance.

Table 12 summarizes the thresholds we need to evaluate *hardware* boundaries, i.e. queue length and utilization bounds. The Th_{maxQL} threshold represents the maximum bound for the hardware device queue length that can be estimated as the average number of all the queue length values, with reference to the entire set of hardware devices in the software system, plus the corresponding variance. Utilization thresholds can be instead refined with ϵ offsets that allow to more precisely determine upper and lower bounds.

Table 13 reports the thresholds we need to evaluate *service* boundaries. In particular, the first column of the Table shows the name of the *threshold* and the second column provides its *description*. For example, the Th_{SthReq} threshold represents the required value for the throughput of the service S. These types of thresholds do not require an heuristic process since we expect that they are defined by software designers in the requirements specification phase.

Table 14 reports the thresholds we need to evaluate operation *slopes* along the time. For example, in the first row the $Th_{OpRtVar}$ threshold represents the maximum feasible slope of the response time observed in n consecutive time slots for

Table 11 Thresholds specification: software characteristics

Threshold	Description	Heuristics
$Th_{maxConnects}$	It represents the maximum bound for the number of usage relationships a software entity is involved	It can be estimated as the average number of usage relationships per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxMsgs}$	It represents the maximum bound for the number of messages sent by a software entity in a service	It can be estimated as the average number of sent messages per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxDBmsgs}$	It represents the maximum bound for the number of database requests in a service	It can be estimated as the average number of database requests per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxRemMsgs}$	It represents the maximum bound for the number of remote messages in a service	It can be estimated as the average number of remote messages per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxRemInst}$	It represents the maximum bound for the number of remote communicating instances in a service	It can be estimated as the average number of remote communicating instances per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
Th_{maxExF}	It represents the maximum bound for the number of exchange formats	It can be estimated as the average number of exchanging formats per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxResDemand}[i]$	It represents the maximum bound for the resource demand of operations	It can be estimated as the average number of resource demands required by a software entity instance, by considering the entire set of software operations in the software system, plus the corresponding variance
$Th_{minResDemand}[i]$	It represents the minimum bound for the resource demand of operations	It can be estimated as the average number of resource demands required by a software entity instance, by considering the entire set of software operations in the software system, minus the corresponding variance
$Th_{maxCrObj}$	It represents the maximum bound for the number of created objects	It can be estimated as the average number of objects dynamically created per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxDeObj}$	It represents the maximum bound for the number of destroyed objects	It can be estimated as the average number of objects dynamically destroyed per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance

Table 12 Thresholds specification: hardware characteristics

Threshold	Description	Heuristics
Th_{maxQL}	It represents the maximum bound for the queue length utilization	It can be estimated as the average number of all hardware devices queue length values, plus the corresponding variance
$Th_{maxHwUtil}$	It represents the maximum bound for the hardware device utilization	It can be estimated as the average number of all hardware devices utilization values, plus the ϵ offset
$Th_{maxNetUtil}$	It represents the maximum bound for the network link utilization	It can be estimated as the average used bandwidth values, with reference to the entire set of network links in the software system, plus the ϵ offset
$Th_{minNetUtil}$	It represents the minimum bound for the network link utilization	It can be estimated as the average used bandwidth values, with reference to the entire set of network links in the software system, minus the ϵ offset
$Th_{maxCpuUtil}$	It represents the maximum bound for the cpu utilization	It can be estimated as the mean cpu devices utilization values, plus the ϵ offset
$Th_{maxDiskUtil}$	It represents the maximum bound for the disk utilization	It can be estimated as the mean disk devices utilization values, plus the ϵ offset
$Th_{minCpuUtil}$	It represents the minimum bound for the cpu utilization	It can be estimated as the mean cpu devices utilization values, minus the ϵ offset
$Th_{minDiskUtil}$	It represents the minimum bound for the disk utilization	It can be estimated as the mean disk devices utilization values, minus the ϵ offset

Table 13 Thresholds specification: requirements

Threshold	Description
Th_{SthReq}	It represents the required value for the throughput of the service S
Th_{SrtReq}	It represents the required value for the response time of the service S

Table 14 Thresholds specification: slopes

Threshold	Description
$Th_{OpRtVar}$	It represents the maximum feasible slope of the response time observed in n consecutive time slots for the operation Op
$Th_{OpThVar}$	It represents the maximum feasible slope of the throughput observed in n consecutive time slots for the operation Op

Table 15 A logic-based representation of Performance Antipatterns

	Antipattern	Formula
Single-value	Circuitous Treasure Hunt	$\exists swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid swE_y.isDB = true \wedge F_{numDBmsgs}(swE_x, swE_y, S) \geq Th_{maxDBmsgs} \wedge F_{maxHwUtil}(P_{swE_y}, all) \geq Th_{maxHwUtil} \wedge F_{maxHwUtil}(P_{swE_y}, disk) > F_{maxHwUtil}(P_{swE_y}, cpu)$
	Blob (or god class/component)	$\exists swE_x, swE_y \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid (F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \vee F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects}) \wedge (F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \vee F_{numMsgs}(swE_y, swE_x, S) \geq Th_{maxMsgs}) \wedge (F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \vee F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil})$
	Unbalanced processing	
	Concurrent Processing Systems	$\exists P_x, P_y \in \mathbb{P} \mid F_{maxQL}(P_x) \geq Th_{maxQL} \wedge [(F_{maxHwUtil}(P_x, cpu) \geq Th_{maxCpuUtil} \wedge F_{maxHwUtil}(P_y, cpu) < Th_{minCpuUtil}) \vee (F_{maxHwUtil}(P_x, disk) \geq Th_{maxDiskUtil} \wedge (F_{maxHwUtil}(P_y, disk) < Th_{minDiskUtil}))]$
	"Pipe and Filter" Architectures	$\exists OpI \in \mathbb{O}, S \in \mathbb{S} \mid \forall i : F_{resDemand}(Op)[i] \geq Th_{resDemand}[i] \wedge F_{probExec}(S, OpI) = 1 \wedge (F_T(S) < Th_{SthReq} \vee F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil})$
	Extensive Processing	$\exists OpI_1, OpI_2 \in \mathbb{O}, S \in \mathbb{S} \mid \forall i : F_{resDemand}(Op_1)[i] \geq Th_{maxOpResDemand}[i] \wedge \forall i : F_{resDemand}(Op_2)[i] < Th_{minOpResDemand}[i] \wedge F_{probExec}(S, OpI_1) + F_{probExec}(S, OpI_2) = 1 \wedge (F_{RT}(S) > Th_{SrtReq} \vee F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil})$
	Empty semi trucks	$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid F_{numRemMsgs}(swE_x, S) \geq Th_{maxRemMsgs} \wedge F_{maxNetUtil}(P_{swE_x}, swE_x) < Th_{minNetUtil} \vee F_{numRemInst}(swE_x, S) \geq Th_{maxRemInst})$
	Tower of Babel	$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid F_{numExF}(swE_x, S) \geq Th_{maxExF} \wedge F_{maxHwUtil}(P_{swE_x}, all) \geq Th_{maxHwUtil}$
	One-Lane Bridge	$\exists swE_x \in \text{sw}\mathbb{E}, S \in \mathbb{S} \mid F_{numSynchCalls}(swE_x, S) \gg F_{poolSize}(swE_x) \wedge F_{serviceTime}(P_{swE_x}) \ll F_{waitingTime}(P_{swE_x}) \wedge F_{RT}(S) > Th_{SrtReq}$
	Excessive dynamic allocation	$\exists S \in \mathbb{S} \mid (F_{numCreatedObj}(S) \geq Th_{maxCrObj} \vee F_{numDestroyedObj}(S) \geq Th_{maxDeObj}) \wedge F_{RT}(S) > Th_{SrtReq}$
Multiple-values	The Ramp	$\exists OpI \in \mathbb{O} \mid \frac{\sum_{1 \leq t \leq n} \lvert (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \rvert}{n} > Th_{OpRtVar} \wedge \frac{\sum_{1 \leq t \leq n} \lvert (F_T(OpI,t) - F_T(OpI,t-1)) \rvert}{n} > Th_{OpThVar}$
	Traffic Jam	$\exists OpI \in \mathbb{O} \mid \frac{\sum_{1 \leq t \leq k} \lvert (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \rvert}{k-1} < Th_{OpRtVar} \wedge \lvert F_{RT}(OpI,k) - F_{RT}(OpI,k-1) \rvert > Th_{OpRtVar} \wedge \frac{\sum_{k \leq t \leq n} \lvert (F_{RT}(OpI,t) - F_{RT}(OpI,t-1)) \rvert}{n-k} < Th_{OpRtVar}$
	More is Less	$\exists P_x \in \mathbb{P} \mid \forall i : F_{par}(P_x)[i] \ll \frac{\sum_{1 \leq t \leq n} (F_{RTpar}(P_x,t)[i] - F_{RTpar}(P_x,t-1)[i])}{n}$

the operation Op. This type of thresholds is necessary for multiple-values antipatterns, and their values depend from the time slot width. This characteristic makes difficult their estimation that should be provided by domain experts.

Finally, Table 15 lists the logic-based representation of the performance antipatterns we propose. Each row represents a specific antipattern reporting its name and the first-order logics *formula* modeling it.

References

1. Document Object Model (DOM) (2000) Java API for XML Processing, Package org.w3c.dom. http://download.oracle.com/javase/1.4.2/docs/api/org/w3c/dom/package-summary.html
2. Jess, the Rule Engine for the Java Platform (2007). http://www.jessrules.com/jess/index.shtml
3. UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group, Inc. (2005). http://www.omg.org/cgi-bin/doc?formal/05-07-04
4. UML Profile for MARTE, OMG document formal/09-11-02, Object Management Group, Inc. (2009). http://www.omg.org/spec/MARTE/1.0/PDF/
5. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: a survey. IEEE Trans. Softw. Eng. **30**(5), 295–310 (2004)
6. Banks, J., Nelson, B.L., Nicol, D.M.: Discrete-Event System Simulation. Prentice Hall, Englewood Cliffs (1999)
7. Bernardi, S., Donatelli, S., Merseguer, J.: From uml sequence diagrams and statecharts to analysable petrinet models. In: WOSP, pp. 35–45 (2002)
8. Boroday, S., Petrenko, A., Singh, J., Hallal, H.: Dynamic analysis of java applications for multithreaded antipatterns. In: Workshop on Dynamic Analysis (WODA), pp. 1–7 (2005)
9. Brown, W.J., Malveau, R.C., McCormick, H.W. III., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, London (1998)
10. Casale, G., Serazzi, G.: Quantitative system evaluation with java modeling tools. In: ICPE'11—Second Joint WOSP/SIPEW International Conference on Performance Engineering, pp. 449–454 (2011)
11. 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)
12. Cortellessa, V., Di Marco, A., Inverardi, P.: Model-Based Software Performance Analysis. Springer, Berlin (2011)
13. Cortellessa, V., Di Marco, A., Trubiani, C.: Performance Antipatterns as Logical Predicates. In: IEEE International Conference on Engineering of Complex Computer Systems, ICECCS, Oxford, UK, pp. 146–156 (2010)
14. Cortellessa, V., Martens, A., Reussner, R., Trubiani, C. : A process to effectively identify "Guilty" performance antipatterns. In: Rosenblum, D.S., Taentzer, G. (eds.) *FASE*, ser. Lecture Notes in Computer Science, vol. 6013, pp. 368–382. Springer, Berlin (2010)
15. Cortellessa, V., Mirandola, R.: Prima-uml: a performance validation incremental methodology on early uml diagrams. Sci. Comput. Program. **44**(1), 101–129 (2002)
16. Dudney, B., Asbury, S., Krozak, J.K., Wittkopf, K.: J2EE Antipatterns. Wiley, London (2003)
17. Elaasar, M., Briand, L.C., Labiche, Y.: A metamodeling approach to pattern specification. In: Lecture Notes in Computer Science: Model Driven Engineering Languages and Systems, Vol. 4199/2006, pp. 484–498. Springer, Berlin (2006)
18. Florescu, O., Voeten, J., Theelen, B., Corporaal, H.: Patterns for automatic generation of soft real-time system models. Simulation **85**(11–12), 709–734 (2009)
19. France, R.B., Kim, D.-K., Ghosh, S., Song, E.: A uml-based pattern specification technique. IEEE Trans. Softw. Eng. **30**(3), 193–206 (2004)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
21. Grunske, L.: Specification patterns for probabilistic quality properties. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE, pp. 31–40. ACM, New York (2008)
22. Harman, M.: The current state and future of search based software engineering. In: Workshop on the Future of Software Engineering (FOSE), pp. 342–357 (2007)
23. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Prentice Hall, Englewood Cliffs (2006)
24. Jain, R.: The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling. SIGMETRICS Perform. Eval. Rev. **19**(2), 5–11 (1991)
25. Kant, K.: Introduction to Computer System Performance Evaluation. McGraw-Hill, New York (1992)
26. Kleinrock, L.: Queueing Systems Vol. 1: Theory. Wiley, London (1975)
27. Kniesel G., Hannemann J., Rho T.: A comparison of logic-based infrastructures for concern detection and extraction. In: Workshop on Linking Aspect Technology and Evolution (2007)
28. Lazowska, E.D., Zahorjan, J., Scott Graham, G., Sevcik, K.C.: Computer System Analysis Using Queueing Network Models. Prentice-Hall, Englewood Cliffs (1984)
29. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: WOSP/SIPEW International Conference on Performance Engineering, pp. 105–116 (2010)
30. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
31. Parsons, T., Murphy, J.: Detecting performance antipatterns in component based enterprise systems. J. Object Technol. **7**(3), 55–90 (2008)
32. Sabetta, A., Petriu, D.C., Grassi, V., Mirandola, R.: Abstraction-raising transformation for generating analysis models. In: MoDELS Satellite Events, pp. 217–226 (2005)
33. Smith, C.U., Williams, L.G.: Software performance antipatterns. In: 2nd International Workshop on Software and Performance, pp. 127–136 (2000)
34. Smith, C.U., Williams, L.G.: Software performance antipatterns; common performance problems and their solutions. In: Int. CMG Conference, pp. 797–806 (2001)
35. Smith, C.U., Williams, L.G.: New software performance antipatterns: More ways to shoot yourself in the foot. In: Computer Measurement Group Conference, pp. 667–674 (2002)
36. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley Longman Publishing Co. Inc., Redwood City (2002)
37. Smith, C.U., Williams, L.G.: More new software performance antipatterns: even more ways to shoot yourself in the foot. In: Computer Measurement Group Conference, pp. 717–725 (2003)
38. Sommerville, I.: Software Engineering, 8th edn. Addison Wesley, Reading (2006)
39. Sup So, S., Deok Cha, S., Rae Kwon, Y.: Empirical evaluation of a fuzzy logic-based software quality prediction model. Fuzzy Sets Syst. **127**(2), 199–208 (2002)
40. Taibi, T., Ngo, D.C.L.: Formal specification of design patterns—a balanced approach. J. Object Technol. **2**(4), 127–140 (2003)
41. Tate, B., Clark, M., Lee, B., Linskey, P.: Bitter EJB. Manning, Shelter Island (2003)
42. Trcka, N., van der Aalst, W.M., Sidorova, N.: Data-flow anti-patterns: discovering dataflow errors in workflows. In: Conference on Advanced Information Systems (CAiSE), vol. 5565, pp. 425–439. LNCS Springer (2009)
43. Trivedi, K.S.: Probability and Statistics with Reliability, Queueing and Computer Science Applications. Wiley, London (2001)
44. Trubiani, C., Koziolek, A.: Detection and solution of software performance antipatterns in palladio architectural models. In: Proceedings of the 2nd ACM/SPEC International Conference on

Performance Engineering (ICPE), pp. 19–30. ACM, New York, USA (2011)
45. Woodside, C.M.: A three-view model for performance engineering of concurrent software. IEEE Trans. Softw. Eng. **21**(9), 754–767 (1995)
46. Woodside, C.M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: WOSP, pp. 1–12 (2005)
47. Extensible Markup Language (XML), World Wide Web Consortium (W3C). http://www.w3.org/XML
48. Xu, J.: Rule-based automatic software performance diagnosis and improvement. In: WOSP, pp. 1–12 (2008)
49. Zheng, T., Woodside, C.M.: Heuristic optimization of scheduling and allocation for distributed systems with soft deadlines. In: Computer Performance Evaluations, Modelling Techniques and Tools, pp. 169–181 (2003)

Author Biographies

Vittorio Cortellessa is an Associate Professor in the Computer Science Department at University of L'Aquila (Italy). Prior joining University of L'Aquila, he has been a post-doc fellow in the European Space Agency (Roma, Italy), a Research Associate in the Computer Science Department at University of Rome "Tor Vergata", and a Research Assistant Professor in the Lane Department of Computer Science and Electrical Engineering at West Virginia University (Morgantown, WV). He has been involved in several research projects in the areas of performance and reliability analysis of software/hardware systems, model-driven engineering and non-functional software validation, which are his current main research interests. He has published about 60 journal and conference articles on these topics. He served and is currently serving in the program committees of conferences in his research areas.

Antinisca Di Marco is Assistant Professor at the University of L'Aquila where she held her Ph.D. in Computer Science in June 2005. Previously she worked as Research Fellow at the University College London, UK, and she collaborated as Post-Doc with the Computer Science Department and with Electronic Engineering Department of the University of Rome "Tor Vergata". Her main research interests include (early) verification and validation of QoS, performance modeling, QoS analysis of autonomic services and context-aware mobile software systems, bio-inspired adaptation mechanisms. She published around 40 journals and conference papers on such topics. She has served as program committee member for several international conferences and workshops, and as reviewer for many journals on her research topics. She has been member of several national and international research projects and, currently, she is involved in the FP7 CONNECT FET and VISION ERC Starting grant (FP7 ideas project).

Catia Trubiani is a Research Fellow at the University of L'Aquila since February 2011. She has received the Ph.D. in Computer Science at the University of L'Aquila (AQ) in April 2011 with a dissertation on the automated generation of architectural feedback from software performance analysis results. Previously she has worked at the Electronic Engineering Department of University of Rome "Tor Vergata" within the framework of the Simple Mobile Services EU FP7 project. She is currently involved in the VISION ERC Starting grant FP7 project. Her main research interests include performance analysis and feedback on software architectures, software performance antipatterns, non-functional trade-off analysis of software systems.