Software Performance Antipatterns: Modeling and Analysis

Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani

University of L'Aquila, Dipartimento di Informatica, Italy
{vittorio.cortellessa,antinisca.dimarco,
catia.trubiani}@univaq.it

Abstract. The problem of capturing performance problems is critical in the software design, mostly because the results of performance analysis (i.e. mean values, variances, and probability distributions) are difficult to be interpreted for providing feedback to software designers. Support to the interpretation of performance analysis results that helps to fill the gap between numbers and design alternatives is still lacking. The aim of this chapter is to present the work that has been done in the last few years on filling such gap. The work is centered on software performance antipatterns, that are recurring solutions to common mistakes (i.e. bad practices) affecting performance. Such antipatterns can play a key role in the software performance domain, since they can be used in the investigation of performance problems as well as in the formulation of solutions in terms of design alternatives.

Keywords: Software Architecture, Performance Evaluation, Antipatterns, Feedback Generation, Design Alternatives.

1 Introduction

In the software development domain there is a very high interest in the early validation of performance requirements because this ability avoids late and expensive fix to consolidated software artifacts.

Model-based approaches, pioneered under the name of Software Performance Engineering (SPE) by Smith [1–3], aim at producing performance models early in the development cycle and using quantitative results from model solutions to refactor the architecture and design [4] with the purpose of meeting performance requirements [5]. Advanced Model-Driven Engineering (MDE) techniques have successfully been used in the last few years to introduce automation in software performance modeling and analysis [6, 7].

Nevertheless, the problem of interpreting the performance analysis results is still quite critical. A large gap exists between the representation of performance analysis results and the feedback expected by software architects. Additionally, the former usually contains numbers (e.g. mean response time, throughput variance, etc.), whereas the latter should embed architectural suggestions, i.e. design

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 290–335, 2012.
© Springer-Verlag Berlin Heidelberg 2012

alternatives, useful to overcome performance problems (e.g. split a software component in two components and re-deploy one of them).

Such activities are today exclusively based on the analysts' experience, and therefore their effectiveness often suffers of lack of automation. MDE techniques represent very promising means on this scenario to tackle the problem.

Figure 1 schematically represents the typical steps that are executed at the architectural phase of the software lifecycle to conduct a model-based performance analysis process. Rounded boxes in the figure represent operational steps whereas square boxes represent input/output data. Vertical lines divide the process in three different phases: in the *modeling* phase, an (annotated) software architectural model is built; in the *performance analysis* phase, a performance model is obtained through model transformation, and such model is solved to obtain the performance results of interest; in the *refactoring* phase, the performance results are interpreted and, if necessary, feedback is generated as refactoring actions on the original software architectural model.

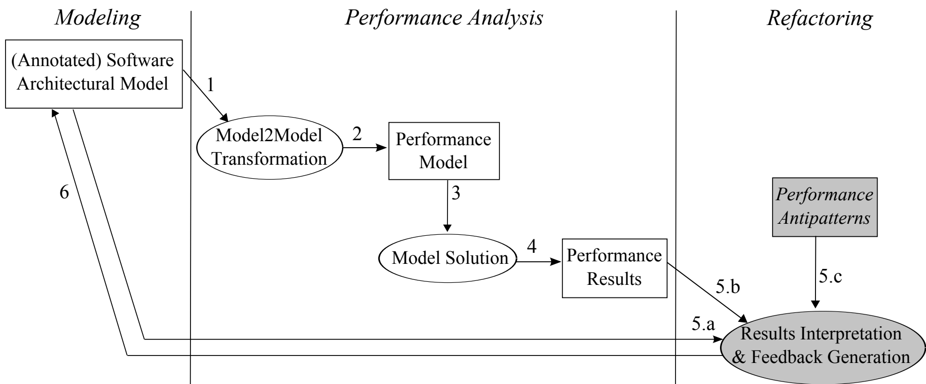


Fig. 1. Model-based software performance analysis process

The modeling and performance analysis phases (i.e. arrows numbered from 1 through 4) represent the forward path from an (annotated) software architectural model all the way through the production of performance indices of interest. As outlined above, while in this path well-founded model-driven approaches have been introduced for inducing automation in all steps (e.g. [6, 8, 9]), there is a clear lack of automation in the backward path that shall bring the analysis results back to the software architecture.

The core step of the backward path is the shaded rounded box of Figure 1. Here, the performance analysis results have to be interpreted in order to detect, if any, performance problems. Once performance problems have been detected (with a certain accuracy) somewhere in the architectural model, solutions have

to be applied to remove those problems[1]. A performance problem originates from a set of unfulfilled requirement(s), such as "the estimated average response time of a service is higher than the required one". If all the requirements are satisfied then the feedback obviously suggests no changes.

In Figure 1 the (annotated) software architectural model (label 5.a) and the performance results (label 5.b) are both inputs to the core step that searches problems in the model. The third input of this step represents the most promising elements that can drive this search, i.e. *performance antipatterns* (label 5.c). The rationale of using performance antipatterns is two-fold: on one hand, a performance antipattern identifies a bad practice in the software architectural model that negatively affects the performance indices, thus it supports the *results interpretation* step; on the other hand, a performance antipattern definition includes a solution description that lets the software architect devise refactoring actions, thus it supports the *feedback generation* step.

The main reference we consider for performance antipatterns is the work done across the years by Smith and Williams [10] that have ultimately defined fourteen notation-independent antipatterns[2]. Some other works present antipatterns that occur throughout different technologies, but they are not as general as the ones defined in [10] (more references are discussed in Section 2 as well as other approaches to the backward path).

Figure 2 details the performance analysis process of Figure 1. In Figure 2, the core step is split in two steps: (i) *detecting antipatterns* that provides the localization of the critical parts of software architectural models, performing the results interpretation step; (ii) *solving antipatterns* that suggests the changes to be applied to the architectural model under analysis, executing the feedback generation step.

Several *iteration*s can be conducted to find the software architectural model that best fits the performance requirements, since several antipatterns may be detected in an architectural model, and several refactoring actions may be available for solving each antipattern. At each iteration, the refactoring actions (labels 6.1 ... 6.h of Figure 2) aim at building a new software architectural model (namely *Candidate*) that replaces the analyzed one. For example, $Candidate_{i-j}$ denotes the j-th candidate generated at the i-th iteration. Then, the detection and solution approach can be iteratively applied to all newly generated candidates to further improve the system, when necessary.

---

[1] Note that this task very closely corresponds to the work of a physician: observing a sick patient (the model), studying the symptoms (some bad values of performance indices), making a diagnosis (performance problem), prescribing a treatment (performance solution through refactoring).

[2] From the original list of fourteen antipatterns [10] two antipatterns are not considered for the following reason: the *Falling Dominoes* antipattern refers to reliability and fault tolerance issues and it is out of interest; the *Unnecessary Processing* antipattern deals with the semantics of the processing by judging the importance of the application code that it is an abstraction level not included in software architectural models. Hence, twelve is the total number of the antipatterns we examine.

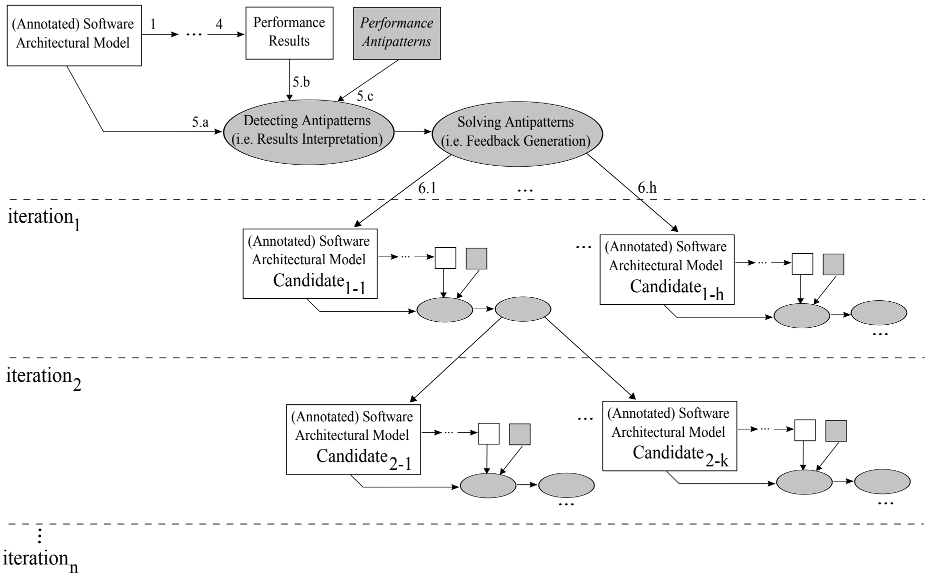


Fig. 2. Software performance analysis process across different iterations

Different termination criteria can be defined in the antipattern-based process: (i) *fulfilment* criterion, i.e. all requirements are satisfied and a suitable software architectural model is found; (ii) *no-actions* criterion, i.e. antipatterns are not detected in the software architectural models therefore no refactoring action can be experimented; (iii) *#iterations* criterion, i.e. the process can be terminated if a certain number of iterations have been completed.

It is worth to notice that the solution of one or more antipatterns does not a priori guarantee performance improvements, because the entire process is based on heuristic evaluations. 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.

The remainder of the chapter is organized as follows. Section 2 discusses existing work in this research area. Sections 3 and 4 present our approach to the representation, and detection/solution activities, needed to embed antipatterns in a software performance process. Section 5 describes a model-driven framework to widen the scope of antipatterns. Finally, Section 6 concludes the chapter by pointing out the pros and cons of using antipatterns in the software performance process and illustrating the most challenging research issues in this area.

2 Related Work

Table 1 summarizes the main existing approaches in literature for the automated generation of architectural feedback. In particular, four categories of approaches are outlined: (i) antipattern-based approaches (see Section 2.1); (ii) rule-based approaches (see Section 2.2); (iii) design space exploration approaches (see Section 2.3); (iv) metaheuristic approaches (see Section 2.4).

Each *approach* is classified on the basis of the category it belongs to. Table 1 compares the different approaches by reporting the *(annotated) software architectural model* and the *performance model* they are based on, if available. The last column of Table 1 denotes as *framework* the set of methodologies the corresponding approach entails. Note that in some cases the framework has been implemented and it is available as a tool (e.g. SPE • ED, ArchE, PerOpteryx).

Our approach somehow belongs to two categories, that are: antipattern-based and rule-based approaches. This is because it makes use of antipatterns for specifying rules that drive towards the identification of performance flaws. *PANDA* (*P*erformance *A*ntipatterns a*N*d Fee*D*back in Software *A*rchitectures) is a framework that embeds all the techniques we propose in this research work that are aimed at performing three main activities, i.e. representing, detecting and solving antipatterns. The implementation of PANDA is still a work in progress and we aim at developing it in the next future.

2.1 Antipattern-Based Approaches

Williams et al. in [11] introduced the PASA (Performance Assessment of Software Architectures) approach. It aims at achieving good performance results through a deep understanding of the architectural features. This is the approach that firstly introduces the concept of antipatterns as support to the identification of performance problems in software architectural models as well as in the formulation of architectural alternatives. However, this approach is based on the interactions between software architects and performance experts, therefore its level of automation is still low.

Cortellessa et al. in [12] introduced a first proposal of automated generation of feedback from the software performance analysis, where performance antipatterns play a key role in the detection of performance flaws. However, this approach considers a restricted set of antipatterns, and it uses informal interpretation matrices as support. The main limitation of this approach is that the interpretation of performance results is only demanded to the analysis of Layered Queue Networks (LQN) [28] performance model. Such knowledge is not enriched with the features coming from the software architectural models, thus to hide feasible refactoring actions.

Enterprise technologies and EJB performance antipatterns are analyzed by Parsons et al. in [13]: antipatterns are represented as sets of rules loaded into an engine. A rule-based performance diagnosis tool, named Performance Antipattern Detection (PAD), is presented. However, it deals with Component-Based Enterprise Systems, targeting only Enterprise Java Bean (EJB) applications.

Table 1. Summary of the approaches for the generation of architectural feedback

	Approach	(Annotated) Software Architectural Model	Performance Model	Framework
Antipattern-based	Williams et al. [11], 2002	Software execution model (Execution graphs)	System execution model (Queueing Network)	SPE • ED
	Cortellessa et al. [12], 2007	Unified Modeling Language (UML)	Layered Queueing Network (LQN)	GARFIELD (Generator of Architectural Feedback through Performance Antipatterns Revealed)
	Parsons et al. [13], 2008	JEE systems from which component level end-to-end run-time paths are collected	Reconstructed run-time design model	PAD (Performance Antipattern Detection)
	Our approach [14] [15], 2009-2011	Unified Modeling Language (UML), Palladio Component Model (PCM)	Queueing Network, Simulation Model	PANDA (Performance Antipatterns aNd FeeDback in Software Architectures)
Rule-based	Barber et al. [16], 2002	Domain Reference Architecture (DRA)	Simulation Model	RARE and ARCADE
	Dobrzanski et al. [17], 2006	Unified Modeling Language (UML)	-	Telelogic TAU (i.e. UML CASE tool)
	McGregor et al. [18], 2007	Attribute-Driven Design (ADD)	Simulation Model	ArchE
	Kavimandan et al. [19], 2009	Real-Time Component Middleware	-	extension of the LwCCM middleware [20]
	Xu [21], 2010	Unified Modeling Language (UML)	Layered Queueing Network (LQN)	PB (Performance Booster)
Design Exploration	Zheng et al. [22], 2003	Unified Modeling Language (UML)	Simulation Model	-
	Bondarev et al. [23], 2007	Robocop Component Model	Simulation model	DeepCompass (Design Exploration and Evaluation of Performance for Component Assemblies)
	Ipek et al. [24], 2008	Artificial Neural Network (ANN)	Simulation Model	-
Metaheuristic	Canfora et al. [25], 2005	Workflow Model	Workflow QoS Model	-
	Aleti et al. [26], 2009	Architecture Analysis and Description Language (AADL)	Markov Model	ArcheOpterix
	Martens et al. [27], 2010	Palladio Component Model (PCM)	Simulation Model	PerOpteryx

From the monitored data of running systems, it extracts the run-time system design and detects EJB antipatterns by applying the defined rules to it. Hence, the scope of [13] is restricted to such domain, and performance problems can neither be detected in other technology contexts nor in the early development stages.

2.2 Rule-Based Approaches

Barber et al. in [16] introduced heuristic algorithms that in presence of detected system bottlenecks provide alternative solutions to remove them. The heuristics are based on architectural metrics that help to compare different solutions. However, it basically identifies and solve only software bottlenecks, more complex problems are not recognized.

Dobrzanski et al. in [17] tackled the problem of refactoring UML models. In particular, *bad smells* are defined as structures that suggest possible problems in the system in terms of functional and non-functional aspects. Refactoring operations are suggested in the presence of bad smells. However, no specific performance issue is analyzed, and refactoring is not driven by unfulfilled requirements.

McGregor et al. in [18] proposed the ArchE framework to support the software designers in creating architectures that meet quality requirements. It embodies knowledge of quality attributes and the relation between the achievement of quality requirements and architectural design. However, the suggestions (or tactics) are not well explained, and it is not clear at which extent the approach can be applied.

Kavimandan et al. in [19] presented an approach to optimize deployment and configuration decisions in the context of distributed, real-time, and embedded component-based systems. Enhanced bin packing algorithms and schedulability analysis have been used to make fine-grained assignments of components to different middleware containers, since they are known to impact on the system performance and resource consumption. However, the scope of this approach is limited to deployment and configuration features.

Xu in [21] presented an approach to software performance diagnosis that identifies performance flaws before the software system implementation. It defines a set of *rules* detecting patterns of interaction between resources. The software architectural models are translated in a performance model, i.e. Layered Queueing Networks (LQNs) [28], and then analyzed. The approach limits the detection to bottlenecks and long execution paths identified and removed at the level of the LQN performance model. The overall approach applies only to LQN models, hence its portability to other notations is yet to be proven and it may be quite complex.

2.3 Design Space Exploration Approaches

Zheng et al. in [22] described an approach to find optimal deployment and scheduling priorities for tasks in a class of distributed real-time systems. In

particular, it is intended to evaluate the deployment of such tasks by applying a heuristic search strategy to LQN models. However, its scope is restricted to adjust the priorities of tasks competing for a processor, and the only refactoring action is to change the allocation of tasks to processors.

Bondarev et al. in [23] presented a design space exploration framework for component-based software systems. It allows an architect to get insight into a space of possible design alternatives with further evaluation and comparison of these alternatives. However, it requires a manual definition of design alternatives of software and hardware architectures, and it is meant to only identify bottlenecks.

Ipek et al. in [24] described an approach to automatically explore the design space for hardware architectures, such as multiprocessors or memory hierarchies. The multiple design space points are simulated and the results are used to train a neural network. Such network can be solved quickly for different architecture candidates and delivers accurate results with a prediction error of less than 5%. However, the approach is limited to hardware properties, it is not suitable for the analysis of software architectural models that usually spread on a wide rage of features.

2.4 Metaheuristic Approaches

Canfora et al. in [25] used genetic algorithms for Quality of Service (QoS)-aware service composition, i.e. to determine a set of *concrete* services to be bound to the *abstract* ones in the workflow of a composite service. However, each basic service is considered as a black-box element, where performance metrics are fixed to certain units, and the genetic algorithms search the best solutions by evaluating the composition options. Hence, no real feedback is given to the designer with the exception of a pre-defined selection of basic services.

Aleti et al. in [26] presented a framework for the optimization of embedded system architectures. In particular, it uses the AADL (Architecture Analysis and Description Language) [29] as the underlying architecture description language and provides plug-in mechanisms to replace the optimization engine, the quality evaluation algorithms and the constraints checking. Architectural models are optimized with evolutionary algorithms considering multiple arbitrary quality criteria. However, the only refactoring action the framework currently allows is the component re-deployment.

Martens et al. in [27] used meta-heuristic search techniques for improving performance, reliability, and costs of component-based software systems. In particular, evolutionary algorithms search the architectural design space for optimal trade-offs by means of Pareto curves. However, this approach is quite time-consuming, because it uses random changes (spanning on all feasible solutions) of the architecture, and the optimality is not guaranteed.

3 Representation of Performance Antipatterns

Performance antipatterns were originally defined in natural language [10]. For sake of simplification, Table 2 reports some examples (i.e. the *Blob*, the *Concurrent Processing Systems*, and the *Empty Semi Trucks* antipatterns [10]) that will be used throughout this section as driving examples. In the table, the *problem* column identifies the system properties that define the antipattern and are useful for detecting it[3]; the *solution* column suggests the architectural changes for solving the antipattern.

Table 2. Some examples of Performance Antipatterns [10]

Antipattern	Problem	Solution
Blob	Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the applications data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the applications top-level classes, and to keep related data and behavior together.
Concurrent Processing Systems	Occurs when processing cannot make use of available processors.	Restructure software or change scheduling algorithms to enable concurrent execution.
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. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.

Starting from their textual description, in the following we provide a graphical representation of performance antipatterns (Section 3.1) in order to quickly convey their basic concepts. The graphical representation (here visualized in a UML-like notation) reflects our interpretation of the textual description of performance antipatterns [10]. It is conceived to capture one reasonable illustration of both the antipattern problem and solution, but it does not claim to be exhaustive. Either the problem or even more the solution description of antipatterns gives rise to a set of options that could be considered to refine the current interpretation.

An antipattern identifies unwanted software and/or hardware properties, thus an antipattern can be formulated as a (maybe complex) logical predicate on the software architectural model elements. In fact, from the informal representation of the *problem* (as reported in Table 2), 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 (Section 3.2).

[3] Such properties refer to software and/or hardware architectural characteristics as well as to the performance indices obtained by the analysis.

3.1 Graphical Representation of Performance Antipatterns

In this section we present the graphical representation of some performance antipatterns, i.e. the *Blob*, the *Concurrent Processing Systems*, and the *Empty Semi Trucks*[4] [10].

We organize the software architectural model elements into views, each capturing a different aspect of the system. Similarly to the Three-View Model [31], 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.

Blob Antipattern

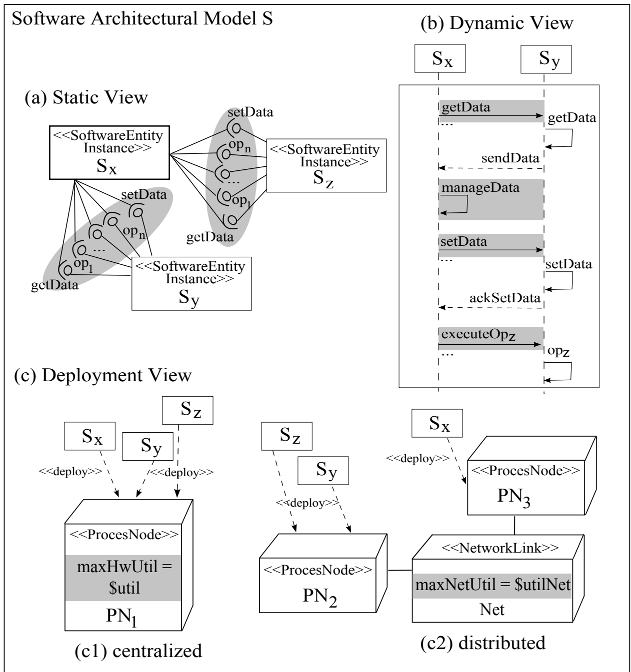
Figures 3 and 4 provide a graphical representation of the *Blob* antipattern in its two forms, i.e. *Blob-controller* and *Blob-dataContainer* respectively.

The upper side of Figures 3 and 4 describes the properties of a *Software Model S* with a *BLOB problem*: (a) *Static View*, a complex software entity instance, i.e. S_x, is connected to other software instances, e.g. S_y and S_z, through *many* dependencies (e.g. *setData*, *getData*, etc.); (b) *Dynamic View*, the software instance S_x generates (see Figure 3) or receives (see Figure 4) *excessive* message traffic to elaborate data managed by other software instances such as S_y; (c) *Deployment View*, it includes two sub-cases: (c1) the centralized case, i.e. if the communicating software instances are deployed on the same processing node then a shared resource will show *high* utilization value, i.e. $\$util$; (c2) the distributed case, i.e. if the communicating software instances are deployed on different processing nodes then the network link will be a critical resource with a *high* utilization value, i.e. $\$utilNet$ [5]. The occurrence of such properties leads to assess that the software resource S_x originates an instance of the Blob antipattern.

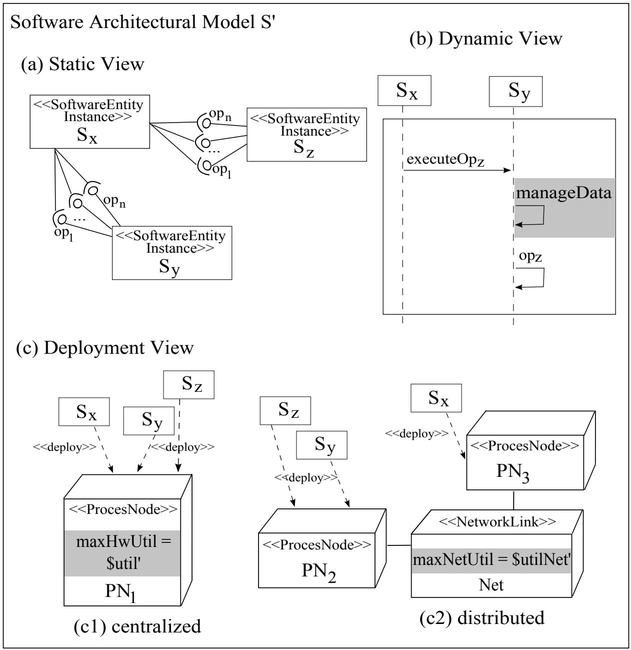
The lower side of Figures 3 and 4 contains the design changes that can be applied according to the *BLOB solution*. The refactoring actions are: (a) the number of dependencies between the software instance S_x and the surrounding ones, like S_y and S_z, must be decreased by delegating some functionalities to the surrounding instances; (b) the number of messages sent (see Figure 3) or received (see Figure 4) by S_x must be decreased by moving the data management from S_x to the surrounding software instances. As consequence of previous actions: (c1) if the communicating software instances were deployed on the same hardware resource then the latter will not be a critical resource anymore, i.e. $\$util' \ll \$util$; (c2) if the communicating software instances are deployed on different

[4] Readers interested to the graphical representation of other antipatterns can refer to [30].

[5] The characterization of antipattern parameters related to system characteristics (e.g. *many* usage dependencies, *excessive* message traffic) or to performance results (e.g. *high*, *low* utilization) is based on thresholds values (see more details in Section 4).

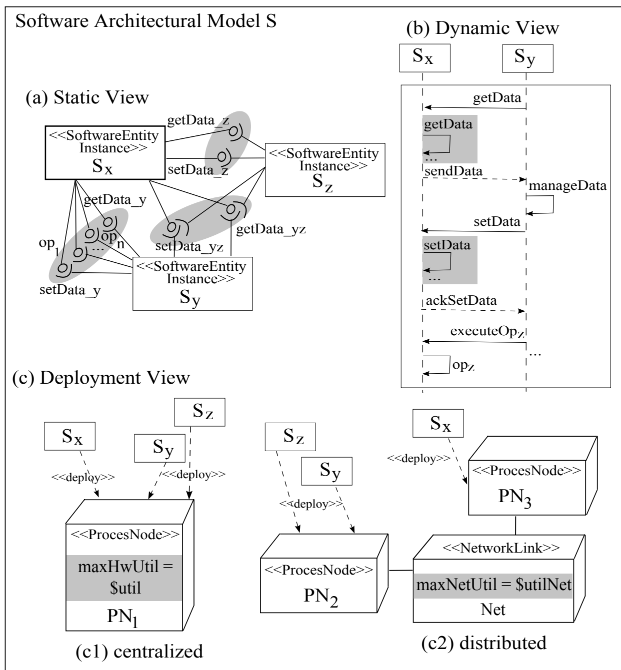


"BLOB-controller" problem

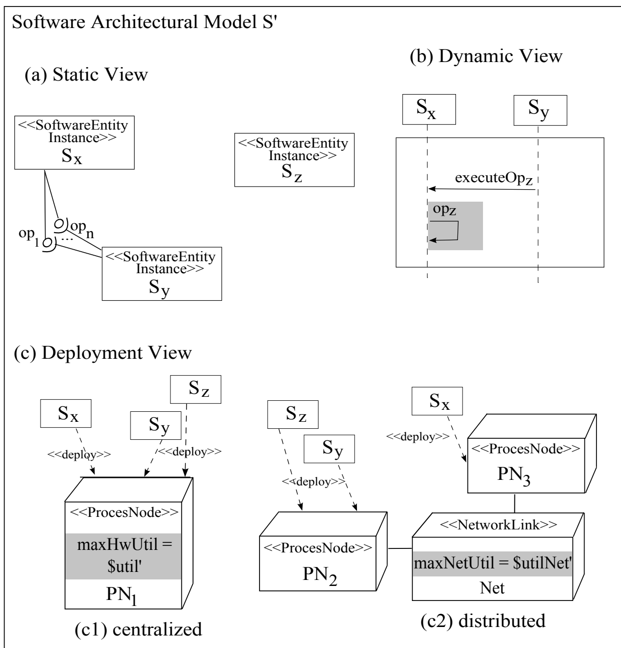


"BLOB-controller" solution

Fig. 3. A graphical representation of the *Blob-controller* Antipattern



"BLOB-dataContainer" problem



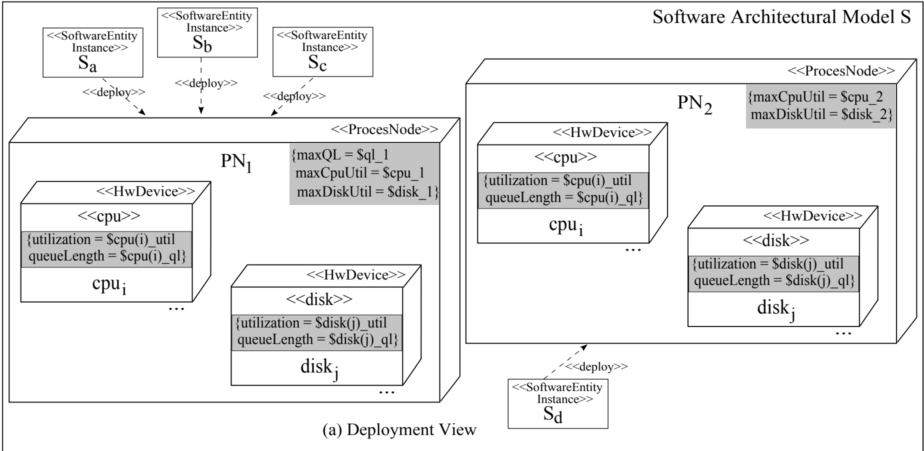
"BLOB-dataContainer" solution

Fig. 4. A graphical representation of the *Blob-dataContainer* Antipattern

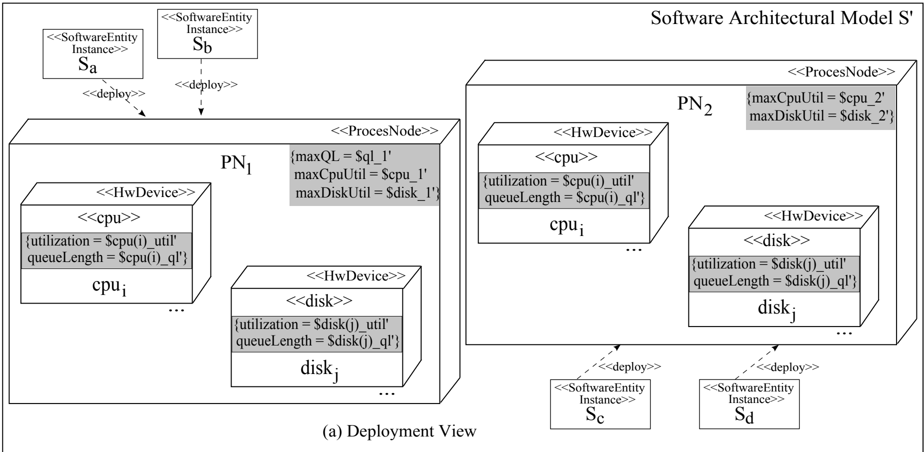
hardware resources then the network will not be a critical resource anymore, i.e. $\$utilNet' \ll \$utilNet$.

Concurrent Processing Systems Antipattern

Figure 5 provides a graphical representation of the *Concurrent Processing Systems* antipattern.



"Concurrent Processing Systems" problem



"Concurrent Processing Systems" solution

Fig. 5. A graphical representation of the *Concurrent Processing Systems* Antipattern

The upper side of Figure 5 describes the system properties of a *Software Model S* with a *Concurrent Processing Systems problem*: (a) *Deployment View*, there are two processing nodes, e.g. PN_1 and PN_2, with un unbalanced processing,

i.e. many tasks are assigned to PN_1 whereas PN_2 is not so heavily used. The over used processing node will show *high* queue length value ($\$ql_1$, estimated as the maximum value overall its hardware devices, i.e. $\$cpu(i)_ql$ and $\$disk(j)_ql$), and a *high* utilization value among its hardware entities either for cpus ($\$cpu_1$, estimated as the maximum value overall its cpu devices, i.e. $\$cpu(i)_util$), and disks ($\$disk_1$, estimated as the maximum value overall its disk devices, i.e. $\$disk(j)_util$). The less used PN_2 processing node will show *low* utilization value among its hardware entities either for cpus ($\$cpu_2$), and disks ($\$disk_2$). The occurrence of such properties leads to assess that the processing nodes PN_1 and PN_2 originate an instance of the Concurrent Processing Systems antipattern.

The lower side of Figure 5 contains the design changes that can be applied according to the *Concurrent Processing Systems solution*. The refactoring actions are: (a) the software entity instances must be deployed in a better way, according to the available processing nodes. As consequences of the previous action, if the software instances are deployed in a balanced way then the processing node PN_1 will not be a critical resource anymore, hence $\$ql_1'$, $\$cpu_1'$, $\$disk_1'$ values improves despite the $\$cpu_2'$, $\$disk_2'$ values.

Empty Semi Trucks Antipattern

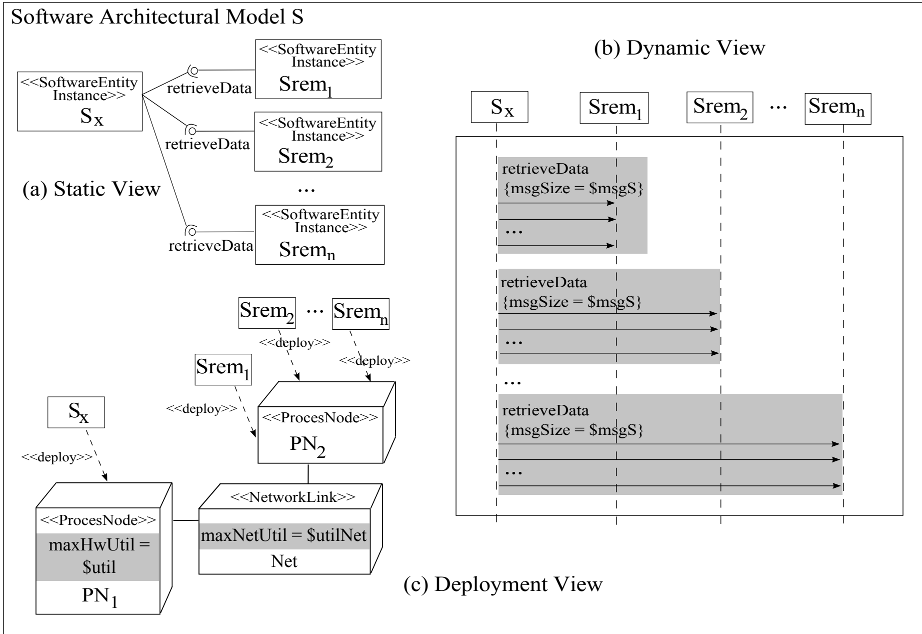
Figure 6 provides a graphical representation of the *Empty Semi Trucks* antipattern.

The upper side of Figure 6 describes the system properties of a *Software Model S* with a *Empty Semi Trucks problem*: (a) *Static View*, there is a software entity instance, e.g. S_x, retrieving some information from several instances ($Srem_1$, $\ldots$, $Srem_n$); (b) *Dynamic View*, the software instance S_x generates an *excessive* message traffic by sending a big amount of messages of *low* sizes ($\$msgS$), much lower than the network bandwidth, hence the network link has a *low* utilization value ($\$utilNet$); (c) *Deployment View*, the processing node on which S_x is deployed, i.e. PN_1, reveals a *high* utilization value ($\$util$). The occurrence of such properties leads to assess that the software instance S_x originates an instance of the Empty Semi Trucks antipattern.

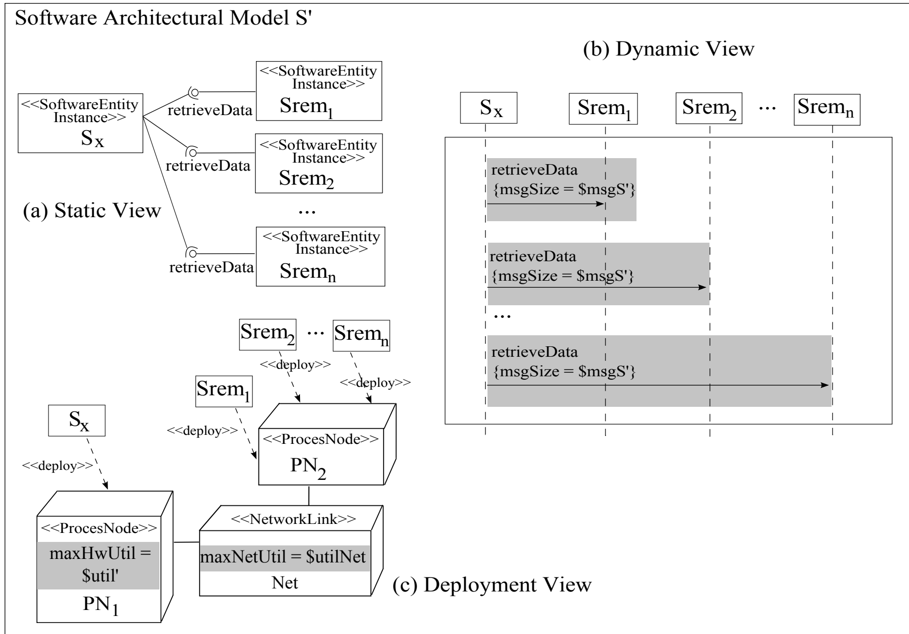
The lower side of Figure 6 contains the design changes that can be applied according to the *Empty Semi Trucks solution*. The refactoring action is: (a) the communication between S_x and the remote instances must be restructured, messages are merged in bigger ones ($\$msgS'$) to reduce the number of messages sent over the network. As consequences of the previous action, if the information is exchanged with a smarter organization of the communication, then the utilization of the processing node hosting S_x is expected to improve, i.e. $\$util' \ll \$util$.

3.2 Logic-Based Representation of Performance Antipatterns

Logical predicates for antipatterns are aimed at defining conditions on specific architectural model elements (e.g. number of interactions among software resources, hardware resources throughput) that we had originally organized in an XML Schema [30], and we here denote with the `typewriter font`.



"Empty Semi Trucks" problem



"Empty Semi Trucks" solution

Fig. 6. A graphical representation of the *Empty Semi Trucks* Antipattern

As shown in Section 3.1, the specification of *model elements* to describe antipatterns is a quite complex task, because such elements can be of different types: (i) elements of a software architectural model (e.g. software resource, message, hardware resource); (ii) performance results (e.g. utilization of a network resource); (iii) structured information that can be obtained by processing the previous ones (e.g. the number of messages sent by a software resource towards another one); (iv) bounds that give guidelines for the interpretation of the system features (e.g. the upper bound for the network utilization).

These two latter model elements, i.e. structured information and bounds, can be defined, respectively, by introducing supporting *functions* that elaborate certain sets of system elements (represented in the predicates as $F_{funcName}$), and *thresholds* that need to be compared with (observed) properties of the software system (represented in the predicates as $Th_{thresholdName}$).

In this section we present the logic-based representation of the performance antipatterns graphically introduced in Section 3.1, that are *Blob*, *Concurrent Processing Systems*, and *Empty Semi Trucks*[6] [10].

Blob Antipattern

The *Blob* (or "god" class/component) antipattern [10] 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. Excessive message traffic that can degrade performance"* (see Table 2).

Following the graphical representation of Figures 3 and 4, 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 `SoftwareEntityInstance`, e.g. swE_x, such that it "*performs all of the work of an application*", while relegating other instances to minor and supporting roles. Let us define by $F_{numClientConnects}$ the function that counts how many times the software entity instance swE_x is in a `Relationship` with other software entity instances by assuming swE_x as client. The property of performing all the work of an application can be checked by comparing the output value of the $F_{numClientConnects}$ function with the $Th_{maxConnects}$ threshold:

$$F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \tag{1}$$

In the second case there is at least one `SoftwareEntityInstance`, e.g. swE_x, such that it "*holds all of the application's data*". 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 swE_x as supplier. The property of holding all of the application's data can be

[6] Readers interested to the logic-based representation of other antipatterns can refer to [30].

checked by comparing the output value of the $F_{numSupplierConnects}$ function with the $Th_{maxConnects}$ threshold:

$$F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects} \tag{2}$$

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 in 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 the $Th_{maxMsgs}$ threshold in both directions:

$$F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \tag{3a}$$

$$F_{numMsgs}(swE_y, swE_x, S) \geq Th_{maxMsgs} \tag{3b}$$

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 $Th_{maxHwUtil}$ threshold. 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 the $Th_{maxHwUtil}$ threshold:

$$F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \tag{4}$$

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 `Proces Node` 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 the $Th_{maxNetUtil}$ threshold:

$$F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil} \tag{5}$$

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

$$\exists swE_x, swE_y \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{((1) \vee (2)) \wedge ((3a) \vee (3b)) \wedge ((4) \vee (5))}$$

where sw$\mathbb{E}$ represents the `SoftwareEntityInstance`s, 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.

Concurrent Processing Systems Antipattern

The *Concurrent Processing Systems* antipattern [10] has the following problem informal definition: *"occurs when processing cannot make use of available processors"* (see Table 2).

Following the graphical representation of Figure 5, 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 greater than the Th_{maxQL} threshold:

$$F_{maxQL}(P_x) \geq Th_{maxQL} \tag{6}$$

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:

$$F_{maxHwUtil}(P_x, cpu) \geq Th_{maxCpuUtil} \tag{7a}$$

$$F_{maxHwUtil}(P_x, disk) \geq Th_{maxDiskUtil} \tag{7b}$$

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:

$$F_{maxHwUtil}(P_y, cpu) < Th_{minCpuUtil} \tag{8a}$$

$$F_{maxHwUtil}(P_y, disk) < Th_{minDiskUtil} \tag{8b}$$

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

$$\exists P_x, P_y \in \mathbb{P} \mid \boxed{(6) \wedge [((7a) \wedge (8a)) \vee ((7b) \wedge ((8b)))]}$$

where $\mathbb{P}$ represents the set of all the `ProcesNode`s 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.

Empty Semi Trucks Antipattern

The *Empty Semi Trucks* antipattern [10] has 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 2).

Following the graphical representation of Figure 6, 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 `Message`s 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. The antipattern can occur when this function returns a value higher or equal than the $Th_{maxRemMsgs}$ threshold:

$$F_{numRemMsgs}(swE_x, S) \geq Th_{maxRemMsgs} \tag{9}$$

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 $Th_{minNetUtil}$ threshold. 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:

$$F_{maxNetUtil}(P_{swE_x}, swE_x) < Th_{minNetUtil} \tag{10}$$

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 the $Th_{maxRemInst}$ threshold:

$$F_{numRemInst}(swE_x, S) \geq Th_{maxRemInst} \tag{11}$$

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

$$\exists swE_x \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid \boxed{(9) \wedge ((10) \vee (11))}$$

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.

Finally, Table 3 lists the logic-based representation of all the performance antipatterns we consider. Each row represents a specific antipattern that is characterized by two attributes: *antipattern* name, and its *formula*, i.e. the first order logics predicate modeling the corresponding antipattern problem.

The list of performance antipatterns has been here enriched with an additional attribute. As shown in the leftmost part of Table 3, 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 simulation or monitoring.

Note that the formalization of antipatterns is the result of multiple formulations and checks. This is a first attempt to formally define antipatterns and it may be subject to some refinements. However, the logic-based formalization was meant to demonstrate the potential for a machine-processable management of performance antipatterns.

4 Detection and Solution of Performance Antipatterns

In this section we apply the antipattern-based approach to an Electronic Commerce System (ECS) case study, modeled with the Unified Modeling Language (UML) [32]. Figure 7 customizes the approach of Figure 1 to the specific methodologies adopted for this case study.

ECS has been modeled with UML annotated with the MARTE profile[7] [33] that provides all the information we need for reasoning on performance issues. The transformation from the software architectural model to the performance model is performed with PRIMA-UML, i.e. a methodology that generates Queueing Network models from UML models [34]. Once the Queueing Network (QN) model is derived, classical QN solution techniques based on well-known methodologies [35], such as Mean Value Analysis (MVA), can be applied to solve it. The performance model is analyzed to obtain the performance indices of interest (i.e. response time, utilization, throughput, etc.).

The UML model and the performance indices are joined in an XML representation[8] of the ECS, parsed by a detection engine that provides the critical

[7] MARTE provides stereotypes and tags to annotate UML models with information required to perform performance analysis.

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

Table 3. A logic-based representation of Performance Antipatterns

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

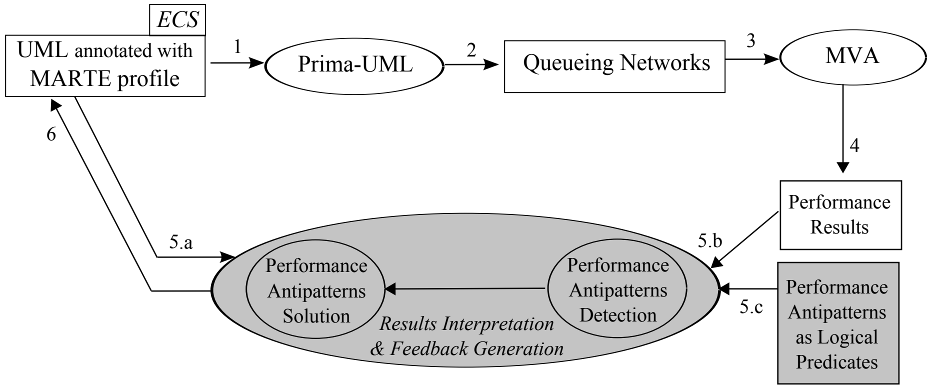


Fig. 7. ECS case study: customized software performance process

elements in architectural models representing the source of performance problems as well as a set of refactoring actions to overcome such issues.

The rest of this section is organized as follows. Section 4.1 describes the UML model of the system under analysis. Then, the stepwise application of our antipattern-based process is performed, i.e. the detection of antipatterns (see Section 4.2) and their solution (see Section 4.3). Finally, in Section 4.4 we briefly discuss a technique to optimize the antipatterns solution process.

4.1 Electronic Commerce System

Figure 8 shows an overview of the ECS software system. It is a web-based system that manages business data: customers browse catalogs and make selections of items that need to be purchased; at the same time, suppliers can upload their catalogs, change the prices and the availability of products, etc. The services we analyze here are *browseCatalog* and *makePurchase*. The former can be perfomance-critical because it is required by a large number of (registered and not registered) customers, whereas the latter can be perfomance-critical because it requires several database accesses that can drop the system performance.

In Figures 9 and 10 we report an excerpt of the ECS annotated software architectural model. We use UML 2.0 [32] as modeling language and MARTE [33] to annotate additional information for performance analysis (such as workload to the system, service demands, hardware characteristics). In particular, the UML Component Diagram in Figure 9 describes the software components and their interconnections, whereas the UML Deployment Diagram of Figure 10 shows the deployment of the software components on the hardware platform. The deployment is annotated with the characteristics of the hardware nodes to specify CPU attributes (*speedFactor* and *schedPolicy*) and network delay (*blockT*).

Performance requirements are defined for the ECS system on the response time of the main services of the system (i.e. *browseCatalog* and *makePurchase*) under a closed workload with a population of 200 requests/second, and thinking

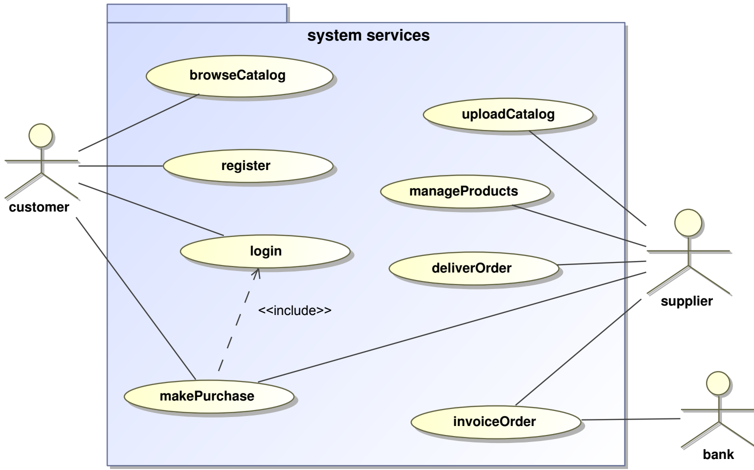


Fig. 8. ECS case study: UML Use Case Diagram

time of 0.01 seconds. The requirements are defined as follows: the *browseCatalog* service must be performed in 1.2 seconds, whereas the *makePurchase* in 2 seconds. These values represent the upper bound for the services they refer to.

The Prima-UML methodology requires the modeling of: (i) system requirements with a UML Use Case Diagram, (ii) the software dynamics with UML Sequence Diagrams, and (iii) the software-to-hardware mapping with a UML Deployment Diagram. The Use Case Diagram must be annotated with the operational profile, the Sequence Diagram with service demands and message size of each operation, and the Deployment Diagram with the characteristics of hardware nodes (see more details in [34]).

Figure 11 shows the Queueing Network model produced for ECS. It 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 Figure 11) is invoked with a probability of 99%, and *makePurchase* (*class B*, denoted with a bullet point in Figure 11) is invoked with a probability of 1%.

The parametrization of the Queueing Network model for the ECS case study is summarized in Table 4. In particular the input parameters of the QN are reported: the first column contains the service center names, the second column shows their corresponding service rates for each class of job (i.e. *class A* and *class B*).

Table 5 summarizes the performance analysis results of the ECS Queueing Network model: the first column contains the names of requirements; the second column reports their *required values*; the third column shows their *predicted values*, as obtained from the QN solution. As it can be noticed both services have a response time that does not fulfill the required ones: the *browseCatalog*

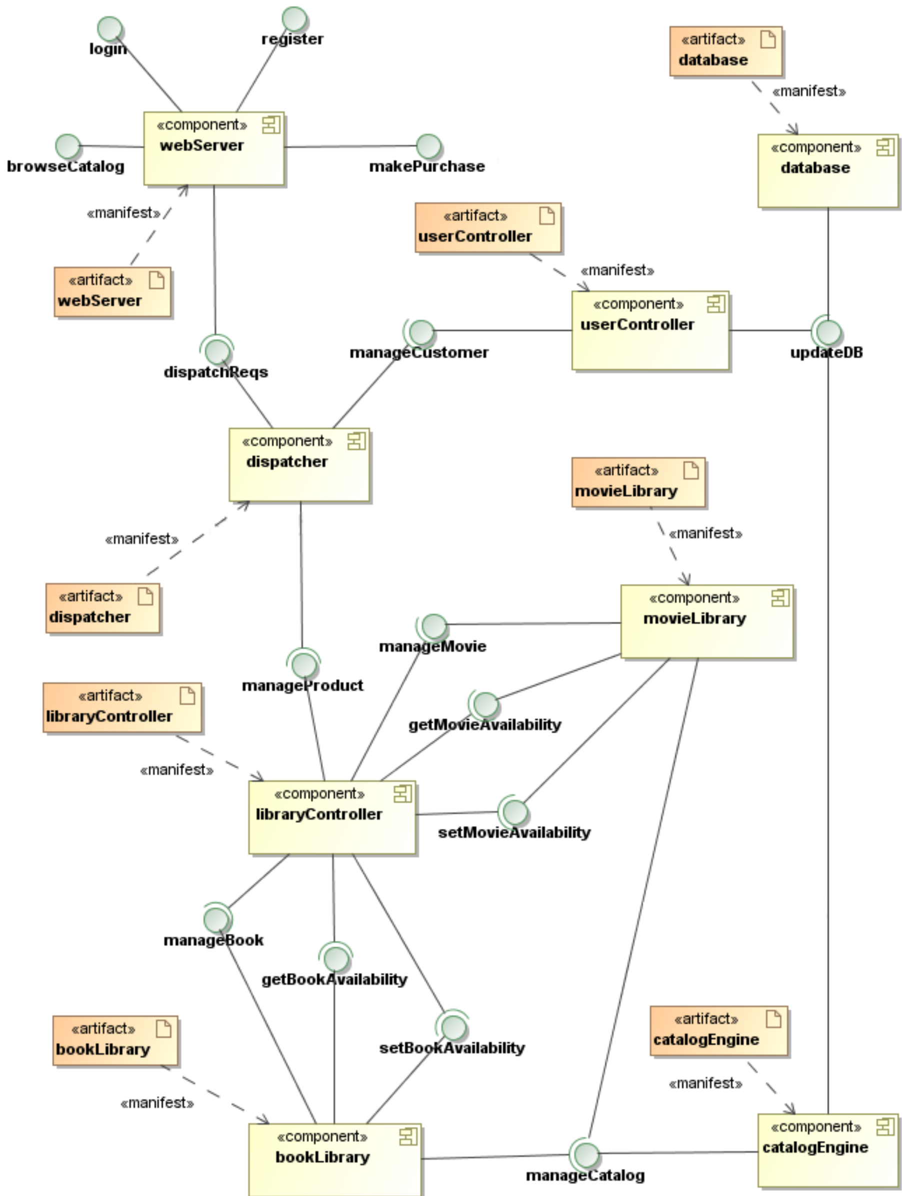


Fig. 9. ECS case study: UML Component Diagram

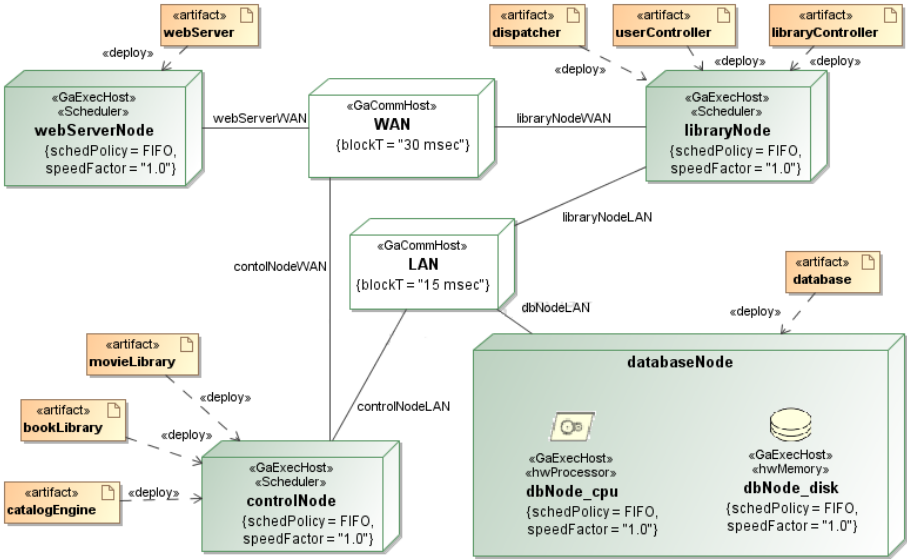


Fig. 10. ECS case study: UML Deployment Diagram

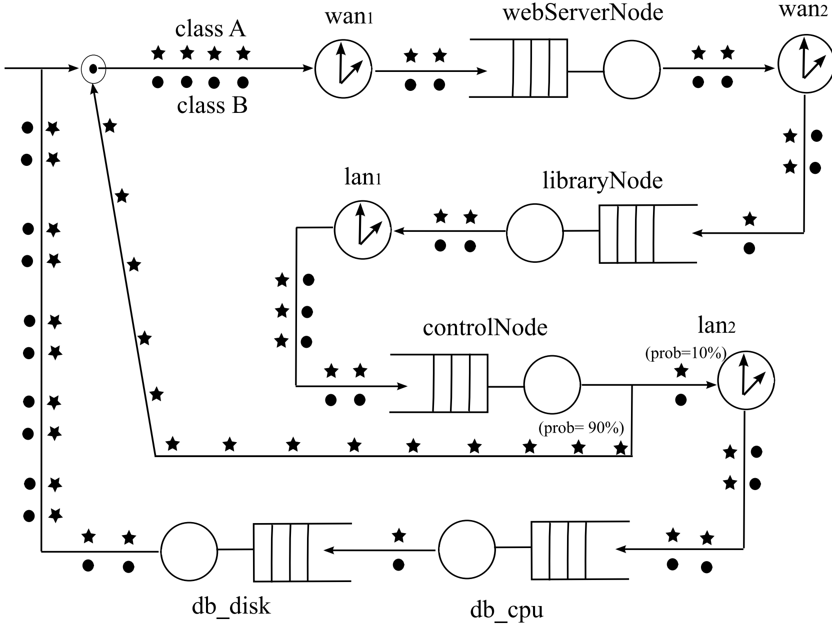


Fig. 11. ECS - Queueing Network model

Table 4. Input parameters for the queueing network model in the ECS system

Service Center	Input parameters ECS	
	$class_A$	$class_B$
lan	44 msec	44 msec
wan	208 msec	208 msec
$webServerNode$	2 msec	4 msec
$libraryNode$	7 msec	16 msec
$controlNode$	3 msec	3 msec
db_cpu	15 msec	30 msec
db_disk	30 msec	60 msec

Table 5. Response time requirements for the ECS software architectural model

Requirement	Required Value	Predicted Value ECS
RT($browseCatalog$)	1.2 sec	1.5 sec
RT($makePurchase$)	2 sec	2.77 sec

service has been predicted as 1.5 sec, whereas the *makePurchase* service has been predicted as 2.77 sec. Hence we apply our approach to detect performance antipatterns.

As said in Section 3.2, basic predicates contain boundaries that need to be actualized on each specific software architectural model. Table 6 reports the

Table 6. ECS- antipatterns boundaries binding

antipattern	parameter	value
Blob	$Th_{maxConnect}$	4
	$Th_{maxMsgs}$	18
	$Th_{maxHwUtil}$	0.75
	$Th_{maxNetUtil}$	0.85
CPS	$Th_{maxQueue}$	40
	$Th_{cpuMaxUtil}$	0.8
	$Th_{diskMaxUtil}$	0.7
	$Th_{cpuMinUtil}$	0.3
	$Th_{diskMinUtil}$	0.4
EST	$Th_{remMsgs}$	12
	$Th_{remInst}$	5
	$Th_{minNetUtil}$	0.3
...	...	...

binding of the performance *antipatterns boundaries* for the ECS system[9]. Such values allow to set the basic predicates, thus to proceed with the actual detection.

4.2 Detecting Antipatterns

The detection of antipatterns is performed by running a detection engine on the XML representation of the ECS software architectural model. This leaded to detect three antipatterns occurrences in the model, that are: Blob, Concurrent Processing Systems, and Empty Semi Trucks.

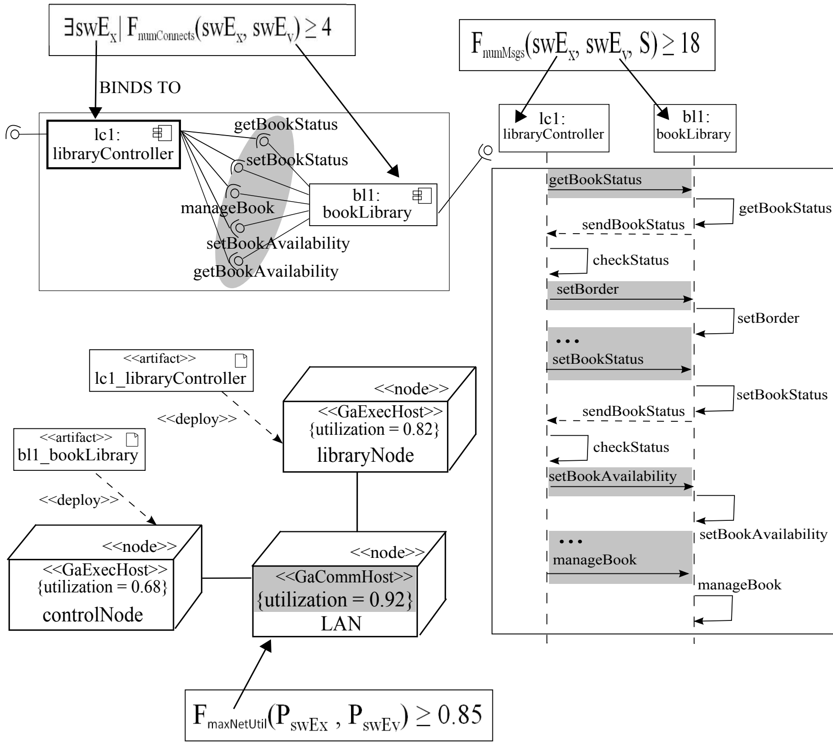


Fig. 12. ECS- the *Blob* antipattern occurrence

In Figure 12 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the parts of the model that give evidence to the Blob antipattern occurrence. Such antipattern is detected since the instance *lc1* of the component *libraryController* satisfies all the Blob logical predicates. In particular (see Table 6 and Figure 12): (a) it has more than

[9] Readers interested to the heuristics used to set antipatterns boundaries can refer to [30].

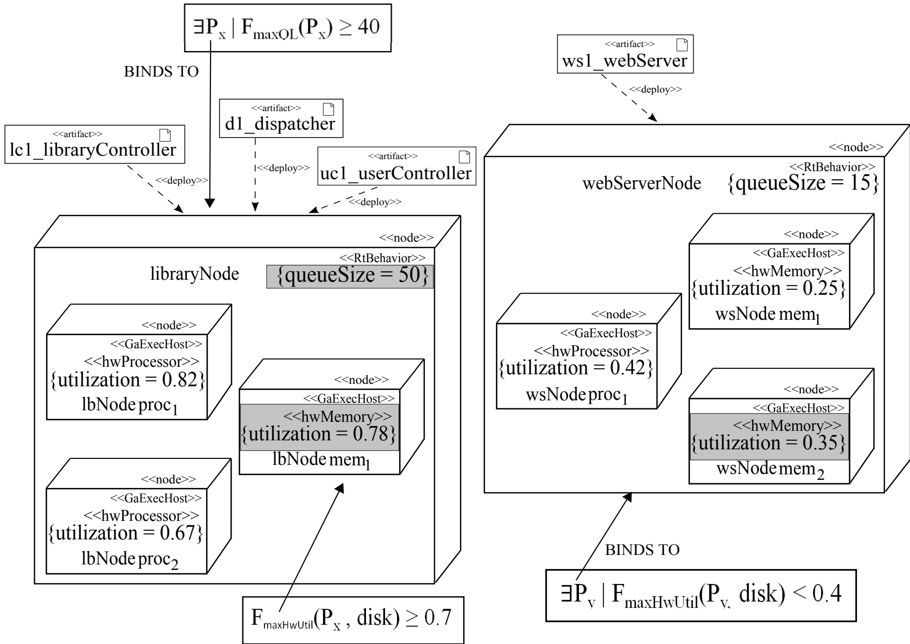


Fig. 13. ECS- the *Concurrent Processing Systems* antipattern occurrence

4 usage dependencies towards the instance *bl1* of the component *bookLibrary*; (b) it sends more than *18* messages (not shown in Figure 12 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*).

In Figure 13 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the parts of the model that give evidence to the CPS antipattern occurrence. Such antipattern is detected the instances *libraryNode* and *webServerNode* satisfy all the CPS logical predeicates. In particular (see Table 6 and Figure 13): (a) the queue size of *libraryNode* (i.e. 50) is higher than the threshold value of *40*; (b) an unbalanced load among CPUs does not occur, because the maximum utilization of CPUs in *libraryNode* (i.e. 0.82 in the $lbNodeproc_1$ instance) is higher than *0.8* threshold value, but the maximum utilization of CPUs in *webServerNode* (i.e. 0.42 in the $wsNodeproc_1$ instance) is not lower than *0.3* threshold value; (c) an unbalanced load among disks occurs, in fact the maximum utilization of disks in *libraryNode* (i.e. 0.78 in the $lbNodemem_1$ instance), is higher than the threshold value of *0.7*, and the maximum utilization of disks in *webServerNode* (i.e. 0.35 in the $wsNodemem_1$ instance), is lower than the threshold value of *0.4*.

In Figure 14 we illustrate an excerpt of the ECS software architectural model where we highlight, in the shaded boxes, the parts of the model that give evidence

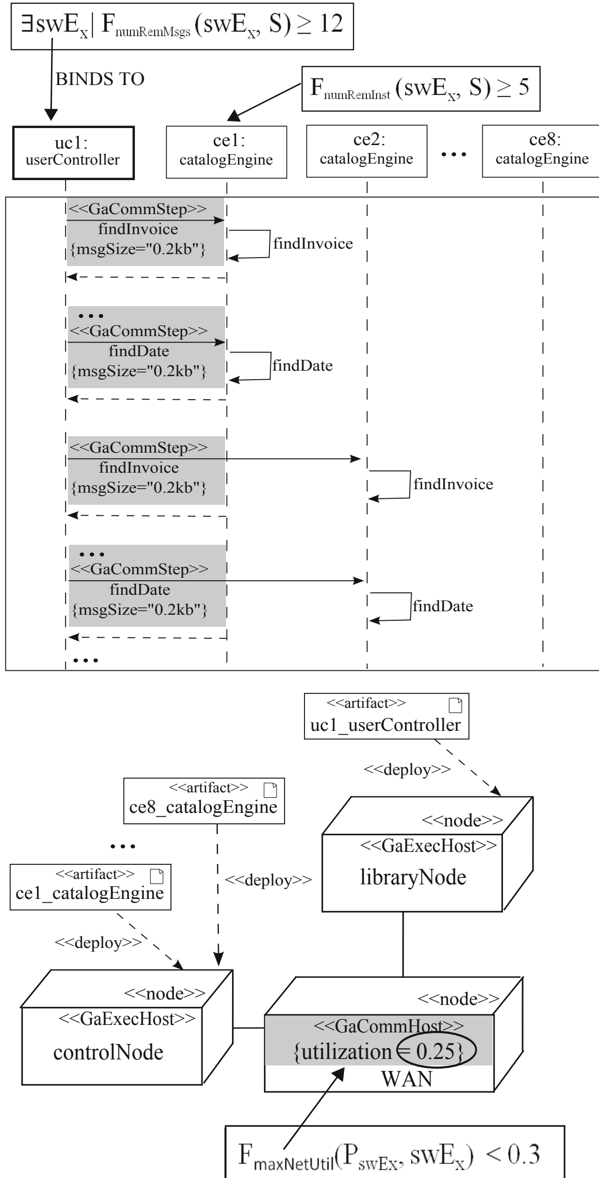


Fig. 14. ECS- the *Empty Semi Trucks* antipattern occurrence

to the EST antipattern occurrence. Such antipattern occurs since the instance *uc1* of the *userController* component satisfies all the EST logical predicates. In particular (see Table 6 and Figure 14): (a) it sends more than *12* remote messages (not shown in Figure 14 for sake of space); (b) the component instances are deployed on different nodes, and the communication host utilization (i.e. 0.25 in the *wan* instance) is lower than the *0.3* threshold value; (c) it communicates with more than *5* remote instances (*ce1*, ..., *ce8*) of the *catalogEngine* component.

4.3 Solving Antipatterns

In Table 7 we have tailored the textual descriptions (see Table 2) on antipattern instances detected on ECS.

Table 7. ECS Performance Antipatterns: problem and solution

Antipattern	Problem	Solution
Blob	*libraryController* performs most of the work, it generates excessive message traffic.	Refactor the design to keep related data and behavior together. Delegate some work from *libraryController* to *bookLibrary*.
Concurrent Processing Systems	Processing cannot make use of the processor *webServerNode*.	Restructure software or change scheduling algorithms between processors *libraryNode* and *webServerNode*.
Empty Semi Trucks	An excessive number of requests is performed for the *makePurchase* service.	Combine items into messages to make better use of available bandwidth.

According to Table 7, we have refactored the ECS (annotated) software architectural model obtaining three new software architectural models, namely $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$, where the Blob, the Concurrent Processing Systems and the Empty Semi Trucks antipatterns have been solved, respectively.

Figure 15 shows the software model $ECS \setminus \{blob\}$ where the *Blob* antipattern is solved by modifying the inner behavior of the *libraryController* software component, thus it delegates some work to the *bookLibrary* component and the logical predicates are not valid anymore. The *Concurrent Processing Systems* antipattern is solved by re-deploying the software component *userController* from *libraryNode* to *webServerNode*. The *Empty Semi Trucks* antipattern is solved by modifying the inner behavior of the *userController* component in the communication with the *catalogEngine* component for the *makePurchase* service.

$ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$ systems have been separately analyzed. Input parameters are reported in Table 8 where bold numbers represent the changes induced from the solution of the corresponding antipatterns.

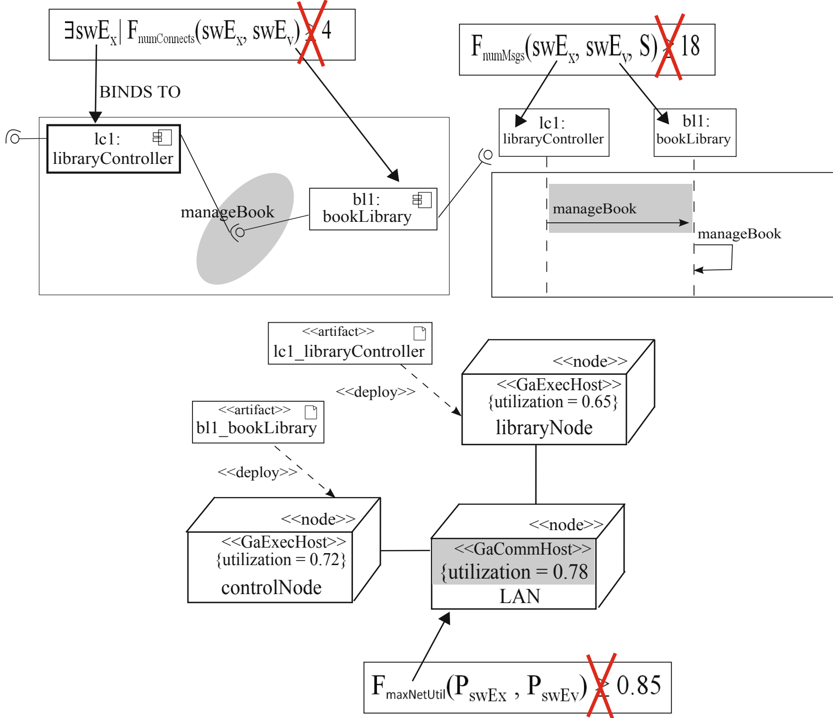


Fig. 15. $ECS \setminus \{blob\}$- the *Blob* antipattern refactoring

For example, in the column $ECS \setminus \{cps\}$ we can notice that the service centers *webServerNode* and *libraryNode* have different input values, since the re-deployment of the software component *userController* implies to move the load from *libraryNode* to *webServerNode*.

In case of *class A*, the load is estimated of 2 msec, in fact in *libraryNode* the initial value of 2 msec in ECS (see Table 4) is increased by 2 msec, thus to become 4 msec in $ECS \setminus \{cps\}$ (see Table 8), whereas in *webServerNode* the initial value of 7 msec in ECS (see Table 4) is decreased by 2 msec, thus to become 5 msec in $ECS \setminus \{cps\}$ (see Table 8). In case of *class B*, the load is estimated of 8 msec, in fact in *libraryNode* the initial value of 4 msec in ECS (see Table 4) is increased by 8 msec, thus to become 12 msec in $ECS \setminus \{cps\}$ (see Table 8), whereas in *webServerNode* the initial value of 16 msec in ECS (see Table 4) is decreased by 8 msec, thus to become 8 msec in $ECS \setminus \{cps\}$ (see Table 8).

Table 9 summarizes the performance analysis results obtained by solving the QN models of the new ECS systems (i.e. $ECS \setminus \{blob\}$, $ECS \setminus \{cps\}$, and $ECS \setminus \{est\}$ columns), and by comparing them with the results obtained from the analysis of the initial system (i.e. ECS column). The response time of the *browseCatalog* service is 1.14, 1.15, and 1.5 seconds, whereas the response time of the *makePurchase* service is 2.18, 1.6, and 2.24 seconds, across the different reconfigurations of the ECS architectural model.

Table 8. Input parameters for the queueing network model across different software architectural models

Service Center	Input parameters					
	$ECS \setminus \{cps\}$		$ECS \setminus \{est\}$		$ECS \setminus \{blob\}$	
	$class_A$	$class_B$	$class_A$	$class_B$	$class_A$	$class_B$
lan	44 msec	44 msec	44 msec	44 msec	44 msec	44 msec
wan	208 msec	208 msec	208 msec	208 msec	208 msec	208 msec
webServerNode	**4 msec**	**12 msec**	2 msec	4 msec	2 msec	4 msec
libraryNode	**5 msec**	**8 msec**	7 msec	**12 msec**	**5 msec**	**14 msec**
controlNode	3 msec	3 msec	3 msec	3 msec	3 msec	3 msec
db_cpu	15 msec	30 msec	15 msec	30 msec	15 msec	30 msec
db_disk	30 msec	60 msec	30 msec	60 msec	30 msec	60 msec

Table 9. Response time required and observed

Requirement	Required Value	Predicted Value			
		ECS	$ECS \setminus \{blob\}$	$ECS \setminus \{cps\}$	$ECS \setminus \{est\}$
RT(*browseCatalog*)	1.2 sec	1.5 sec	1.14 sec	1.15 sec	1.5 sec
RT(*makePurchase*)	2 sec	2.77 sec	2.18 sec	1.6 sec	2.24 sec

The solution of the Blob antipattern satisfies 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.

We can conclude that the software architectural model candidate that best fits with user needs is obtained by applying the following refactoring action: the *userController* software component is re-deployed from *libraryNode* to *webServerNode*, i.e. the solution of the Concurrent Processing Systems antipattern. In fact, as shown in Table 9 both requirements have been fulfilled by its solution, i.e. the *fulfilment* termination criterion (see Section 1). The experimental results are promising, and other decisions can be taken by looking at these results, as opposite to the common practice where software architects use to blindly act without this type of information.

4.4 A Step Ahead in the Antipatterns Solution

In this section the problem of identifying, among a set of detected performance antipattern instances, the ones that are the real causes of problems (i.e. the "guilty" ones) is tackled. In particular, it is introduced a process to elaborate the performance analysis results and to score performance requirements, model entities and performance antipattern instances. The cross observation of such scores allows to classify the level of guiltiness of each antipattern.

Figure 16 reports the process that we propose: the goal is to modify a software architectural model in order to produce a model *candidate* where the performance problems of the former have been removed. Shaded boxes of Figure 16 represent the *ranking step* that is object of this section.

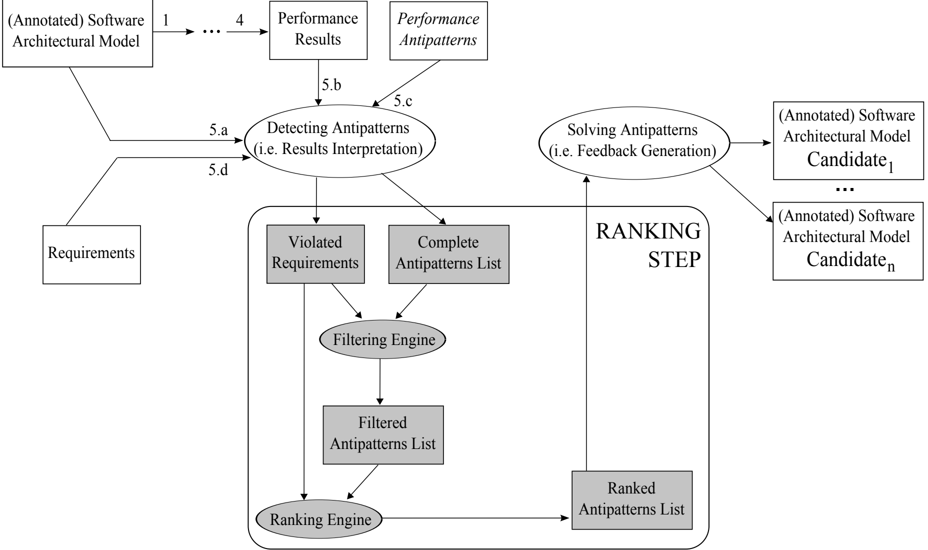


Fig. 16. A process to improve the performance analysis interpretation

The typical inputs of the detection engine are: the software architectural model, the performance results, and the performance antipatterns representation (see Figure 1). We here also report performance *requirements* (label 5.d) because they will be used in the ranking step. We obtain two types of outputs from the detection step: (i) a list of *violated requirements* as resulting from the analysis, and (ii) a *complete antipatterns list*. If no requirement is violated by the current software architectural model then the process terminates.

Then we compare the complete antipatterns list with the violated requirements and examine relationships between detected antipattern instances and each violated requirement through the system entities involved in them. We obtain a *filtered antipatterns list*, where instances that do not affect any violated requirement have been filtered out.

On the basis of relationships observed before, we estimate how guilty an antipattern instance is with respect to a violated requirement by calculating a guiltiness score. As a result, we obtain a *ranked antipatterns list* for each violated requirement. Finally, *candidate*s software architectural model can be obtained by applying the solutions of one or more high-ranked antipattern instances to the current software architectural model for each violated requirement[10].

[10] For sake of space we do not detail this approach here, but interested readers can refer to [36].

5 Plugging Antipatterns in a Model-Driven Framework

In this section we discuss the problem of interpreting the performance analysis results and generating architectural feedback by means of a model-driven framework that supports the antipatterns management. The aim is to make use of all basic and advanced model-driven techniques.

We recall that the main activities performed within such framework are: (i) representing antipatterns (see Section 5.1), to define in a well-formed way the properties that lead the software system to reveal a bad practice, as well as the changes that provide a solution; (i) detecting and solving antipatterns (see Section 5.2), to actually locate and remove antipatterns in software models. Finally, Section 5.3 provides some afterthoughts about the model-driven framework.

5.1 Model-Driven Representation of Antipatterns

The activity of representing antipatterns is performed on this framework by introducing a metamodel (i.e. a neutral and a coherent set of interrelated concepts) to collect the system elements that occur in the definition of antipatterns (e.g. software entity, network resource utilization, etc.), which is meant to be the basis for a machine-processable definition of antipatterns.

This section briefly presents the metamodel, named Performance Antipattern Modeling Language (PAML), that collects all the system elements identified by analyzing the antipatterns definition in literature [10].

The PAML structure is shown in Figure 17. It is constituted of two main parts as delimited by the horizontal dashed line: (i) the *Antipattern Specification* collects the high-level features, such as the views of the system (i.e. static, dynamic, deployment) and their boolean relationships; (ii) the *Model Elements Specification* collects the concepts of the software architectural models and the performance results.

All the architectural model elements and the performance indices occurring in antipatterns' specifications are grouped in a metamodel called SML+ (see Figure 17). SML+ shares many concepts with existing Software Modeling Languages. However, it is not meant to be another modeling language, rather it is oriented to specify the basic elements of performance antipatterns[11].

An antipattern can be specified as a PAML-based model that is intended to formalize its textual description (similarly to what we have done with the logic-based representation of Section 3.2). For example, following the graphical representation of the Blob antipattern (see Figure 3), the corresponding PAML-based model will be constituted by an `AntipatternSpecification` with three `AntipatternViews`: (a) the `StaticView`, (b) the `DynamicView`, (c) the `DeploymentView` for which two `AntipatternSubViews` are defined, i.e. (c1) the centralized one and (c2) the distributed one. A `BooleanRestriction` can be defined

[11] For sake of space we do not detail SML+ here. However, a restricted set of model elements, such as software entity, processing node, etc., are shown in Figure 18, and readers interested to the whole language can refer to [30].

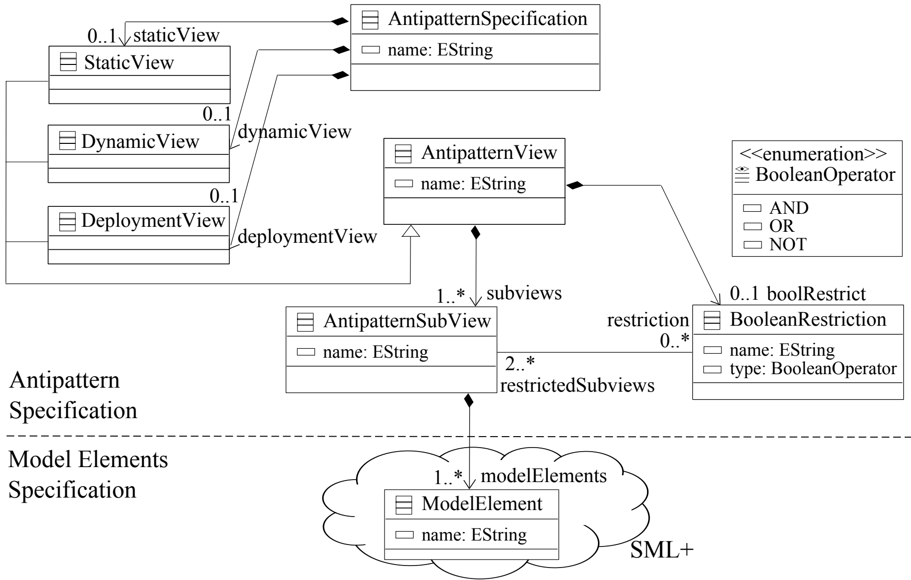


Fig. 17. The Performance Antipattern Modeling Language (PAML) structure

between these sub-views, and the *type* is set by the `BooleanOperator` equal to the *OR* value. Each subview will contain a set of `ModelElements`.

5.2 Model-Driven Detection and Solution of Antipatterns

The activities of detecting and solving antipatterns are performed on this framework by translating the antipatterns representation into concrete modeling notations. In fact, the modeling language used for the target system, i.e. the (annotated) software architectural model of Figure 1, is of crucial relevance, since the antipatterns neutral concepts must be translated into the actual concrete modeling language, if possible[12].

Our model-driven framework is currently considering two concrete notations: UML [32] plus MARTE profile [33]; and the Palladio Component Model (PCM) [37]. Note that the subset of target modeling languages is being enlarged (e.g. with an Architecture Description Language like Æmilia [38]) as far as the concepts for representing antipatterns are available.

Figure 18 shows how the neutral specification of performance antipatterns in PAML can be translated into concrete modeling languages. In fact, antipatterns are built on a set of model elements belonging to SML+, i.e. the infrastructure upon which constructing the semantic relations among different notations.

The semantic relations between a concrete modeling language and SML+ depend on the expressiveness of the target modeling language. For example, in

[12] It depends on the expressiveness of the target modeling language.

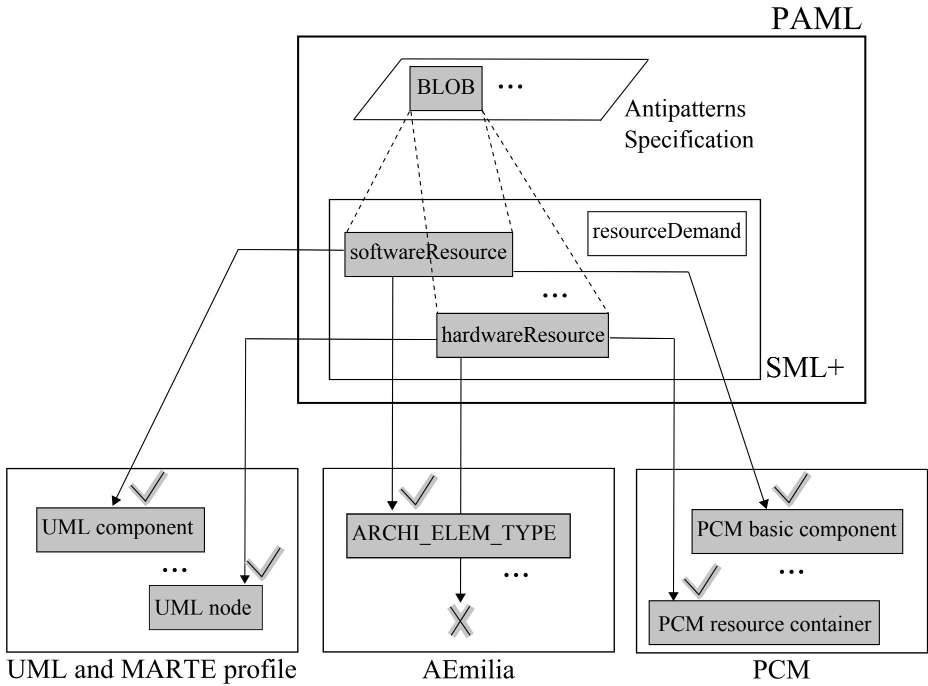


Fig. 18. Translating antipatterns into concrete modeling languages

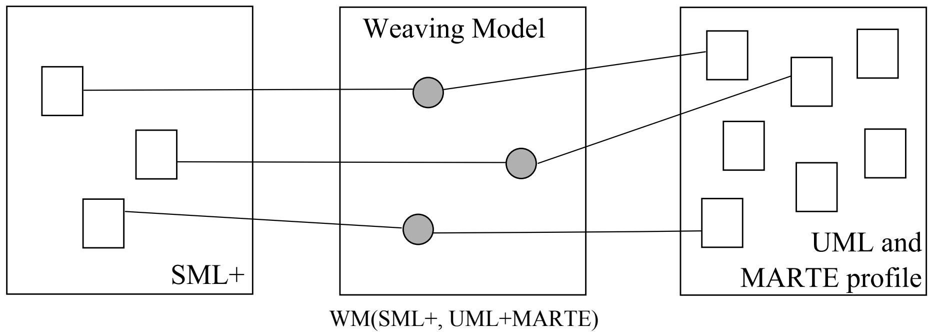


Fig. 19. Metamodel instantiation via weaving models

Figure 18 we can notice that a *SoftwareEntity* is respectively translated in a *UML Component*, a *PCM Basic Component*, and an *Æmilia ARCHI_ELEM_TYPE*. On the contrary, the *ProcesNode* translation is only possible to a *UML Node* and a *PCM Resource Container*, whereas in Æmilia this concept remains uncovered.

We can therefore assert that in a concrete modeling language there are antipatterns that can be automatically detected (i.e. when the entire set of SML+

model elements can be translated in the concrete modeling language) and other ones that are not detectable (i.e. when a restricted set of model elements is translated).

Weaving models [39] can be defined by mapping the concepts of SML+ into the corresponding concepts of a concrete modeling language (as done in [40] for different purposes, though). Weaving models represent useful instruments in software modeling, as they can be used for setting fine-grained relationships between models or metamodels and for executing operations based on the link semantics.

Figure 19 depicts how weaving models define the correspondences among two metamodels, hence the concepts in SML+ can be mapped on those of a concrete notation (e.g. UML and MARTE profile).

The benefit of weaving models is that they can be used in automated transformations to generate other artifacts. In fact it is possible to define high-order transformations that, starting from the weaving models, can generate metamodel-specific transformations that allow to embed the antipatterns in an actual concrete modeling language.

Figure 20 shows how to automatically generate antipatterns models in concrete modeling languages with the usage of weaving models. The metamodel we propose for antipatterns is PAML containing SML+ (box $PAML[SML{+}]_{MM}$). Performance antipatterns are defined as models conform to the PAML metamodel (box $PAML[SML{+}]_{M}$). Antipatterns models in concrete modeling languages can be automatically generated by using the high-order transformation T that takes as input the weaving model WM specifying correspondences between *SML+* and a concrete notation (e.g. *UML+MARTE*) metamodel. Hence, performance antipatterns in UML+MARTE are defined as models (box $PAML[UML{+}MARTE]_{M}$) conform to the PAML metamodel containing UML+MARTE (box $PAML[UML + MARTE]_{MM}$).

A first experimentation in this setting has been conducted in UML+MARTE where antipatterns can be naturally expressed by means of OCL [41] expressions, i.e. model queries with diagrammatic notations that correspond to first-order predicates.

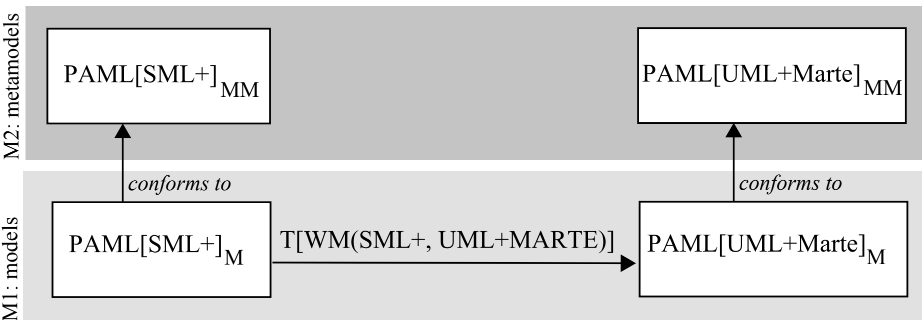


Fig. 20. Weaving model over different software modeling languages

Figure 21 shows the process that gives to each antipattern model an OCL-based semantics, in a similar way to [42], and OCL detection code can be automatically generated from the antipattern specification. The leftmost part of the Figure 21 reports again the PAML metamodel (box $PAML[UML + MARTE]_{MM}$) and antipatterns models (box $PAML[UML + MARTE]_{M}$) defined in the UML+MARTE concrete modeling notation. Firstly, antipatterns models are translated into intermediate models (box OCL_M) conforming to the OCL metamodel (box OCL_{MM}) with a model-to-model transformation. The *OCL code* is generated by using a model-to-text transformation, and it is used to check software model elements, thus to actually perform the antipattern detection. Note that the PAML metamodel provides semantics in terms of OCL: a semantic anchoring [43] is realized by means of automated transformations that map each antipattern model to an OCL expression.

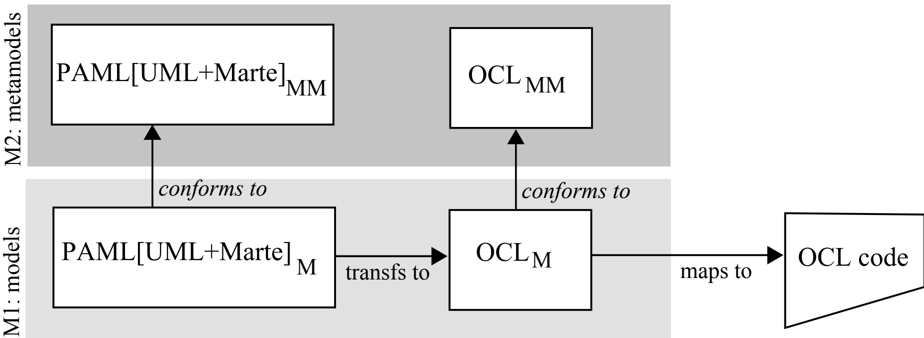


Fig. 21. Tranforming PAML-based models in OCL code

5.3 Afterthoughts

The benefits of introducing a metamodel for representing antipatterns are manifold: (i) *expressiveness*, as it currently contains all the concepts needed to specify performance antipatterns introduced in [10]; (ii) *usability*, as it allows a user-friendly representation of (existing and upcoming) performance antipatterns; (iii) *extensibility*, i.e., if new antipatterns are based on additional concepts the metamodel can be extended to introduce such concepts.

Note that the set of the antipatterns can be enlarged as far as the concepts for representing new ones are available. Technology-specific antipatterns, such as EJB and J2EE antipatterns [44] [45], can be also suited to check if the current metamodel is reusable in domain-specific fields. For example, we retain that the EJB Bloated Session Bean Antipattern [44] can be currently specified as a PAML-based model, since it describes a situation in EJB systems where a session bean has become too bulky, thus it is very similar to the Blob antipattern in the Smith-Williams' classification.

Currently PAML only formalizes the performance problems captured by antipatterns. As future work we plan to complete PAML with a Refactoring Modeling Language (RML) for formalizing the solutions in terms of refactorings,

i.e. changes of the original software architectural model. Such formalization may be supported by high order transformations (similarly to what done for problems) that express the refactoring in concrete modeling languages.

The problem of refactoring architectural models is intrinsically complex and requires specialized algorithms and notations to match the abstraction level of models [46]. Recently, in [47, 48] two similar techniques have been introduced to represent refactorings as difference models. Interestingly these proposals combine the advantages of declarative difference representations and enable the reconstruction of the final model by means of automated transformations which are inherently defined in the approaches.

6 Discussion and Conclusions

In this chapter we dealt with the automated generation of performance feedback in software architectures. We devised a methodology to keep track of the performance knowledge that usually tends to be fragmented and quickly lost, with the purpose of interpreting the performance analysis results and suggesting the most suitable architectural refactoring. Such knowledge base is aimed at integrating different forms of data (e.g. architectural model elements, performance indices), in order to support relationships between them and to manage the data over time, while the development advances.

The performance knowledge that we have organized for reasoning on performance analysis results can be considered as an application of data mining to the software performance domain. It has been grouped around design choices and analysis results concepts, thus to act as a data repository available to reason on the performance of a software system. Performance antipatterns have been of crucial relevance in this context since they represent the source of the concepts to identify performance flaws as well as to provide refactorings in terms of architectural alternatives.

6.1 Summary of Contributions

A list of the main scientific contributions is given in the following.

Specifying Performance Antipatterns. The activity of *specifying antipatterns* has been addressed in [15]: a structured description of the system elements that occur in the definition of antipatterns has been provided, and performance antipatterns have been modeled as logical predicates. Additionally, in [15] the operational counterpart of the antipattern declarative definitions as logical predicates has been implemented with a java rule-engine application. Such engine was able to detect performance antipatterns in an XML representation of the software system that grouped the software architectural model and the performance results data.

A Model-Driven Approach for Antipatterns. A Performance Antipattern Modeling Language (PAML), i.e. a metamodel specifically tailored to describe

antipatterns, has been introduced in [14]. Antipatterns are represented as PAML-based models allows to manipulate their (neutral) specification. In fact in [14, 49] it has been also discussed a vision on how model-driven techniques (e.g. weaving models [39], difference models [47]) can be used to build a notation-independent approach that addresses the problem of embedding antipatterns knowledge across different modeling notations.

Detecting and Solving Antipatterns in UML and PCM. The activities of *detecting* and *solving antipatterns* have been currently implemented by defining the antipattern rules and actions into two modeling languages: (i) the UML and MARTE profile notation in [50]; (ii) the PCM notation in [51]. In [50] performance antipatterns have been automatically detected in UML models using OCL [41] queries, but we have not yet automated their solution. In [51] a limited set of antipatterns has been automatically detected and solved in PCM models through a benchmark tool. These experiences led us to investigate the expressiveness of UML and PCM modeling languages by classifying the antipatterns in three categories: (i) detectable and solvable; (ii) semi-solvable (i.e. the antipattern solution is only achieved with refactoring actions to be manually performed); (iii) neither detectable nor solvable.

A Step Ahead in the Antipatterns Solution. Instead of blindly moving among the antipattern solutions without eventually achieving the desired results, a technique to rank the antipatterns on the basis of their guiltiness for violated requirements has been defined in [52] [36], thus to decide how many antipatterns to solve, which ones and in what order. Experimental results demonstrated the benefits of introducing ranking techniques to support the activity of solving antipatterns.

6.2 Open Issues and Future Work

There are several open issues in the current version of the framework and many directions can be identified for future work.

6.2.1 Short/Medium Term Issues

Further Validation. The approach has to be more extensively validated in order to determine the extent to which it can offer support to user activities. The validation of the approach includes two dimensions: (i) it has to be exposed to a set of target users, such as graduate students in a software engineering course, model-driven developers, more or less experienced software architects, in order to analyze its scope and usability; (ii) it has to be applied to complex case studies by involving industry partners, in order to analyze its scalability. Such experimentation is of worth interest because the final purpose is to integrate the framework in the daily practices of the software development process.

Both the detection and the solution of antipatterns generate some pending issues that give rise to short term goals.

The detection of antipatterns presents the following open issues:

Accuracy of Antipatterns Instances. The detection process may introduce false positive/negative instances of antipatterns. We outlined some sources to suitably tune the values of antipatterns boundaries, such as: (i) the system requirements; (ii) the domain expert's knowledge; (iii) the evaluation of the system under analysis. However, threshold values inevitably introduce a degree of uncertainty and extensive experimentation must be done in this direction. Some fuzziness can be introduced for the evaluation of the threshold values [53]. It might be useful to make antipattern detection rules more flexible, and to detect the performance flaws with higher/lower accuracy.

Some metrics are usually used to estimate the efficiency of design patterns detection, such as *precision* (i.e. measuring what fraction of detected pattern occurrences are real) and *recall* (i.e. measuring what fraction of real occurrences are detected). Such metrics do not apply for antipatterns because usually negative patterns are not explicitly documented in projects' specifications, due to their nature of revealing bad practices. A confidence value can be associated to an antipattern to quantify the probability that the formula occurrence corresponds to the antipattern presence.

Relationship between Antipatterns Instances. The detected instances might be related to each other, e.g. one instance can be the generalization or the specialization of another instance. A dependence value can be associated to an antipattern to quantify the probability that its occurrence is dependent from other antipatterns presence.

The solution of antipatterns presents the following open issues:

No Guarantee of Performance Improvements. The solution of one or more antipatterns does not guarantee performance improvements in advance: the entire process is based on heuristics evaluations. Applying a refactoring action results in a new software architectural model, i.e. a candidate whose performance analysis will reveal if the action has been actually beneficial for the system under study. However, an antipattern-based refactoring action is usually a correctness-preserving transformation that does not alter the semantics of the application, but it may improve the overall performance.

Dependencies of Performance Requirements. The application of antipattern solutions leads the system to (probably) satisfy the performance requirements covered by such solutions. However, it may happen that a certain number of other requirements get worse. Hence, the new candidate architectural model must take into account at each stage of the process all the requirements, also the previously satisfied ones.

Conflict between Antipattern Solutions. The solution of a certain number of antipatterns cannot be unambiguously applied due to incoherencies among their solutions. It may happen that the solution of one antipattern suggests to split a component into three finer grain components, while another antipattern at the same time suggests to merge the original component with another one.

These two actions obviously contradict each other, although no pre-existing requirement limits their application. Even in cases of no explicit conflict between antipattern solutions, coherency problems can be raised from the order of application of solutions. In fact the result of the sequential application of two (or more) antipattern solutions is not guaranteed to be invariant with respect to the application order. Criteria must be introduced to drive the application order of solutions in these cases. An interesting possibility may be represented by the *critical pairs analysis* [54] that provides a mean to avoid conflicting and divergent refactorings.

6.2.2 Long Term Issues

Lack of Model Parameters. The application of the antipattern-based approach is not limited (in principle) along the software lifecycle, but it is obvious that an early usage is subject to lack of information because the system knowledge improves while the development process progresses. Both the architectural and the performance models may lack of parameters needed to apply the process. For example, internal indices of subsystems that are not yet designed in details cannot be collected. Lack of information, or even uncertainty, about model parameter values can be tackled by analyzing the model piecewise, starting from sub-models, thus to bring insight on the missing parameters.

Influence of Domain Features. Different cross-cutting concerns such as the workload, the operational profile, etc. usually give rise to different performance analysis results that, in turn, may result in different antipatterns identified in the system. This is a critical issue and, as usually in performance analysis experiments, the choice of the workload(s) and operational profile(s) must be carefully conducted.

Influence of Other Software Layers. We assume that the performance model only takes into account the (annotated) software architectural model that usually contains information on the software application and hardware platform. Between these two layers there are other components, such as different middlewares and operating systems, that can embed performance antipatterns. The approach shall be extended to these layers for a more accurate analysis of the system. An option can be to integrate benchmarks or models suitable for these layers in our framework.

Limitations from Requirements. The application of antipattern solutions can be restricted by functional or non-functional requirements. Example of functional requirements may be legacy components that cannot be split and redeployed whereas the antipattern solution consists of these actions. Example of non-functional requirements may be budget limitations that do not allow to adopt an antipattern solution due to its extremely high cost. Many other examples can be provided of requirements that (implicitly or explicitly) may affect the antipattern solution activity. For sake of automation such requirements should

be pre-defined so that the whole process can take into account them and preventively excluding infeasible solutions.

Consolidated Formalization of Performance Antipatterns. The Performance Antipatterns Modeling Language (PAML) currently only formalizes the performance problems captured by antipatterns. As future work we plan to complete PAML with a Refactoring Modeling Language (RML) for formalizing the solutions in terms of refactorings, i.e. changes of the original software architectural model.

Note that the formalization of antipatterns reflects our interpretation of the informal literature. Different formalizations of antipatterns can be originated by laying on different interpretations. This unavoidable gap is an open issue in this domain, and certainly requires a wider investigation to consolidate the formal definition of antipatterns. Logical predicates of antipatterns can be further refined by looking at probabilistic model checking techniques, as experimented in [55].

Architectural Description Languages. The framework is currently considering two modeling notations: UML and PCM. In general, the subset of target modeling languages can be enlarged as far as the concepts for representing antipatterns are available; for example, architectural description languages such as AADL [56] can be also suited to validate the approach. A first investigation has been already conducted on how to specify, detect, and solve performance antipatterns in the Æmilia architectural language [38], however it still requires a deep experimentation.

Multi-objective Goals. The framework currently considers only the performance goals of software systems. It can be extended to other quantitative quality criteria of software architectures such as reliability, security, etc., thus to support trade-off decisions between multiple quality criteria.

Acknowledgments. This work has been partially supported by VISION ERC project (ERC-240555).

References

1. Smith, C.U., Millsap, C.V.: Software performance engineering for oracle applications: Measurements and models. In: Int. CMG Conference, Computer Measurement Group, pp. 331–342 (2008)
2. Williams, L.G., Smith, C.U.: Software performance engineering: A tutorial introduction. In: Int. CMG Conference, Computer Measurement Group, pp. 387–398 (2007)
3. Smith, C.U.: Introduction to Software Performance Engineering: Origins and Outstanding Problems. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 395–428. Springer, Heidelberg (2007)
4. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Software Eng. 30, 126–139 (2004)

5. Woodside, C.M., Franks, G., Petriu, D.C.: The Future of Software Performance Engineering. In: Briand, L.C., Wolf, A.L. (eds.) FOSE, pp. 171–187 (2007)
6. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Trans. Software Eng. 30, 295–310 (2004)
7. Cortellessa, V., Di Marco, A., Inverardi, P.: Model-Based Software Performance Analysis. Springer (2011)
8. Koziolek, H.: Performance evaluation of component-based software systems: A survey. Perform. Eval. 67, 634–658 (2010)
9. 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. ACM (2005)
10. Smith, C.U., Williams, L.G.: More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In: International Computer Measurement Group Conference, pp. 717–725 (2003)
11. Williams, L.G., Smith, C.U.: PASA(SM): An Architectural Approach to Fixing Software Performance Problems. In: International Computer Measurement Group Conference, Computer Measurement Group, pp. 307–320 (2002)
12. Cortellessa, V., Frittella, L.: A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis. In: Wolter, K. (ed.) EPEW 2007. LNCS, vol. 4748, pp. 171–185. Springer, Heidelberg (2007)
13. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. Journal of Object Technology 7, 55–91 (2008)
14. Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., Trubiani, C.: Approaching the Model-Driven Generation of Feedback to Remove Software Performance Flaws. In: EUROMICRO-SEAA, pp. 162–169. IEEE Computer Society (2009)
15. Cortellessa, V., Di Marco, A., Trubiani, C.: Performance Antipatterns as Logical Predicates. In: Calinescu, R., Paige, R.F., Kwiatkowska, M.Z. (eds.) ICECCS, pp. 146–156. IEEE Computer Society (2010)
16. Barber, K.S., Graser, T.J., Holt, J.: Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation. In: ASE, pp. 172–182. IEEE Computer Society (2002)
17. Dobrzanski, L., Kuzniarz, L.: An approach to refactoring of executable UML models. In: Haddad, H. (ed.) ACM Symposium on Applied Computing (SAC), pp. 1273–1279. ACM (2006)
18. McGregor, J.D., Bachmann, F., Bass, L., Bianco, P., Klein, M.: Using arche in the classroom: One experience. Technical Report CMU/SEI-2007-TN-001, Software Engineering Institute, Carnegie Mellon University (2007)
19. Kavimandan, A., Gokhale, A.: Applying Model Transformations to Optimizing Real-Time QoS Configurations in DRE Systems. In: Mirandola, R., Gorton, I., Hofmeister, C. (eds.) QoSA 2009. LNCS, vol. 5581, pp. 18–35. Springer, Heidelberg (2009)
20. Object Management Group (OMG): Lightweight CCM RFP. OMG Document realtime/02-11-27 (2002)
21. Xu, J.: Rule-based automatic software performance diagnosis and improvement. Perform. Eval. 67, 585–611 (2010)
22. Zheng, T., Woodside, M.: Heuristic Optimization of Scheduling and Allocation for Distributed Systems with Soft Deadlines. In: Kemper, P., Sanders, W.H. (eds.) TOOLS 2003. LNCS, vol. 2794, pp. 169–181. Springer, Heidelberg (2003)
23. Bondarev, E., Chaudron, M.R.V., de Kock, E.A.: Exploring performance trade-offs of a JPEG decoder using the deepcompass framework. In: International Workshop on Software and Performance, pp. 153–163 (2007)

24. Ipek, E., McKee, S.A., Singh, K., Caruana, R., de Supinski, B.R., Schulz, M.: Efficient architectural design space exploration via predictive modeling. ACM Transactions on Architecture and Code Optimization (TACO) 4 (2008)
25. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Beyer, H.G., O'Reilly, U.M. (eds.) GECCO, pp. 1069–1075. ACM (2005)
26. Aleti, A., Björnander, S., Grunske, L., Meedeniya, I.: ArcheOpterix: An extendable tool for architecture optimization of AADL models. In: ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pp. 61–71 (2009)
27. 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)
28. Petriu, D.C., Shen, H.: Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 159–177. Springer, Heidelberg (2002)
29. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis and Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-001, Software Engineering Institute, Carnegie Mellon University (2006)
30. Trubiani, C.: Automated generation of architectural feedback from software performance analysis results. PhD thesis, University of L'Aquila (2011)
31. Woodside, C.M.: A Three-View Model for Performance Engineering of Concurrent Software. IEEE Transactions on Software Engineering (TSE) 21, 754–767 (1995)
32. Object Management Group (OMG): UML 2.0 Superstructure Specification. OMG Document formal/05-07-04 (2005)
33. Object Management Group (OMG): UML Profile for MARTE. OMG Document formal/08-06-09 (2009)
34. Cortellessa, V., Mirandola, R.: PRIMA-UML: a performance validation incremental methodology on early UML diagrams. Sci. Comput. Program. 44, 101–129 (2002)
35. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. SIGMETRICS Performance Evaluation Review 19, 5–11 (1991)
36. Cortellessa, V., Martens, A., Reussner, R., Trubiani, C.: A Process to Effectively Identify "Guilty" Performance Antipatterns. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 368–382. Springer, Heidelberg (2010)
37. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82, 3–22 (2009)
38. Bernardo, M., Donatiello, L., Ciancarini, P.: Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. In: Calzarossa, M.C., Tucci, S. (eds.) Performance 2002. LNCS, vol. 2459, pp. 236–260. Springer, Heidelberg (2002)
39. Bézivin, J.: On the unification power of models. Software and System Modeling 4, 171–188 (2005)
40. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A.: Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. IEEE Trans. Software Eng. 36, 119–140 (2010)
41. Object Management Group (OMG): OCL 2.0 Specification. OMG Document formal/2006-05-01 (2006)

42. Stein, D., Hanenberg, S., Unland, R.: A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 77–92. Springer, Heidelberg (2005)
43. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
44. Dudney, B., Asbury, S., Krozak, J.K., Wittkopf, K.: J2EE Antipatterns (2003)
45. Tate, B., Clark, M., Lee, B., Linskey, P.: Bitter EJB (2003)
46. Lin, Y., Zhang, J., Gray, J.: Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In: OOPSLA Workshop on Best Practices for Model-Driven Software Development (2004)
47. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology 6, 165–185 (2007)
48. Rivera, J.E., Vallecillo, A.: Representing and Operating with Model Differences. In: International Conference on TOOLS, pp. 141–160 (2008)
49. Trubiani, C.: A Model-Based Framework for Software Performance Feedback. In: Dingel, J., Solberg, A. (eds.) MODELS 2010 Workshops. LNCS, vol. 6627, pp. 19–34. Springer, Heidelberg (2011)
50. 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)
51. Trubiani, C., Koziolek, A.: Detection and solution of software performance antipatterns in palladio architectural models. In: International Conference on Performance Engineering (ICPE), pp. 19–30 (2011)
52. Cortellessa, V., Martens, A., Reussner, R., Trubiani, C.: Towards the identification of "Guilty" performance antipatterns. In: WOSP/SIPEW International Conference on Performance Engineering, pp. 245–246 (2010)
53. So, S.S., Cha, S.D., Kwon, Y.R.: Empirical evaluation of a fuzzy logic-based software quality prediction model. Fuzzy Sets and Systems 127, 199–208 (2002)
54. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci. 127, 113–128 (2005)
55. Grunske, L.: Specification patterns for probabilistic quality properties. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE, pp. 31–40. ACM (2008)
56. Feiler, P.H., Lewis, B.A., Vestal, S.: SAE, Architecture Analysis and Design Language (AADL), as5506/1 (2006), http://www.sae.org