Guilt-based Handling of Software Performance Antipatterns in Palladio Architectural Models

Catia Trubiani[a], Anne Koziolek[c], Vittorio Cortellessa[b], Ralf Reussner[c]

[a] *Gran Sasso Science Institute, L'Aquila, Italy*
[b] *University of L'Aquila, L'Aquila, Italy*
[c] *Karlsruhe Institute of Technology, Karlsruhe, Germany*

Abstract

Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems. Software Performance Antipatterns document common performance problems in the design as well as their solutions. The definition of performance antipatterns concerns software properties that can include static, dynamic, and deployment aspects. To make use of such knowledge, we propose an approach that helps software architects to identify and solve performance antipatterns. Our approach provides software performance feedback to architects, since it suggests the design alternatives that allow overcoming the detected performance problems.

The feedback process may be quite complex since architects may have to assess several design options before achieving the architectural model that best fits the end-user expectations. In order to optimise such process we introduce a ranking methodology that identifies, among a set of detected antipatterns, the "guilty" ones, i.e. the antipatterns that more likely contribute to the violation of specific performance requirements. The introduction of our ranking process leads the system to converge towards the desired performance improvement by discarding a consistent part of design alternatives. Four case studies in different application domains have been used to assess the validity of the approach.

Keywords: Software Performance Engineering, Software Performance Antipatterns, Architectural Feedback, Model-based Performance Analysis, Palladio Architectural Models.

1. Introduction

Software performance is a pervasive quality difficult to model, because it is affected by many aspects of the design and execution environment. A promising trend in this domain is an automatic performance optimisation of architecture, design and run-time configuration [1]: the model and measurement information will be fed back into the software design, so that performance issues are tackled early in the design process.

Figure 1 schematically represents the typical steps to run a complete software performance modelling and analysis process in the software life-cycle. Ellipses in the figure represent operational steps whereas square boxes represent input/output data. Dashed lines divide the process in four different phases: in the *requirements* phase, a set of performance requirements are defined; in the *modelling* phase, software architects build an (annotated[1]) software model; in the *analysis* phase, a performance model is obtained through model-to-model transformations, and such model is solved to obtain the performance indices of interest; in the *refactoring* phase, the performance indices are interpreted and, if necessary, feedback is generated as refactoring actions on the original software model.

The modelling and analysis phases have been quite successfully addressed in the last decade by several approaches that have introduced automation in all steps (e.g. [2, 3, 4]). There is, instead, a clear lack of automation in the refactoring phase, which shall improve

Email addresses: catia.trubiani@gssi.infn.it (Catia Trubiani), koziolek@kit.edu (Anne Koziolek), vittorio.cortellessa@univaq.it (Vittorio Cortellessa), reussner@kit.edu (Ralf Reussner)

[1] In order to conduct quantitative performance analysis, a software model must be extended with performance annotations such as the workload of the system, service demands of operational steps, hardware characteristics, etc.

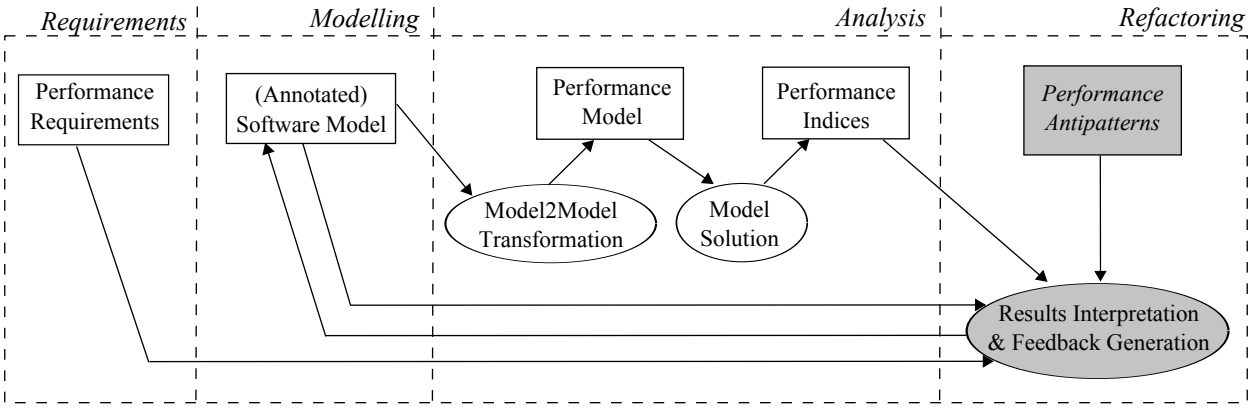

Figure 1: Software performance modelling and analysis process.

the software architecture based on the analysis results. The goal of the refactoring phase, whose core step is the *results interpretation and feedback generation* (see Figure 1), is to look for performance flaws (i.e. not fulfilled performance requirements) in the software model and to provide architectural alternatives[2]. Such activities are today exclusively based on the analysts' experience, and therefore their effectiveness often suffers the lack of automation.

In this work we intend to support the idea that the localization and removal of performance flaws can greatly benefit from an automated detection and solution of performance antipatterns.

Performance antipatterns [5] are descriptions of problems commonly encountered by performance engineers in practice and represent promising instruments to introduce automation in the refactoring phase. The benefit of using antipatterns is two-fold: on the one hand, a performance antipattern identifies a bad practice in the software model that negatively affects the performance indices, thus supporting the *results interpretation* activity; on the other hand, a performance antipattern definition includes a solution description that lets the architect devise refactoring actions, thus supporting the *feedback generation* activity.

This paper is an extension of [6], where we presented the first approach to automatically detect and solve performance antipatterns in a design-level software modelling language. We examined performance antipatterns within the Palladio Component Model (PCM) [7], which is a domain specific modelling language to describe component-based software architectures.

[2]It is obvious that if all performance requirements are satisfied then the feedback simply suggests no change on the software model.

The generic software performance process of Figure 1 is instantiated, for this paper context, in Table 1: the requirements (e.g. the response time or throughput for services) are specified using the Quality of service Modeling Language (QML) [8] as described in [9]; the software system is modelled with PCM; the transformation of the software model into the performance model generates the simulation code for the PCM simulation tool SimuCom [7]; the performance model is then simulated to obtain the performance indices of interest (e.g. response time, utilisation, throughput).

Table 1: This paper context.

Generic process	**This paper context**
Requirements	QML
(Annotated) Software Model	PCM
Model2Model Transformation	PCM2SimuCom
Performance Model	Extended Queueing Network (G/G/n queues + routing + passive resources)
Model Solution	SimuCom simulation
Performance Indices	Response time, Utilisation, Throughput, ...
Performance Antipatterns	Rules and Actions on PCM model elements
Results Interpretation & Feedback Generation	Detection, Ranking and Solution of Antipatterns in PCM

The bottommost entries of Table 1 highlight the insertion of performance antipatterns knowledge in the software performance cycle: a set of *rules* and *actions* are defined in terms of the PCM meta-model elements. On the basis of [10], each rule characterizes the properties to detect performance antipatterns in the PCM model under analysis, whereas each action describes the changes to solve antipatterns in such model.

The set of refactoring actions able to overcome per-

formance problems is the result of multiple experimentations. It may happen that several antipatterns are detected and many design options are defined for the solution of each antipattern. Furthermore, each design option (e.g. the re-deployment of a software component) gives rise to an alternative software model that must be evaluated, and only after the performance analysis has been conducted it is possible to assess the improvements of such alternative.

In this paper we introduce a *ranking* process to support the antipatterns' solution step, as reported in the last row of Table 1. Once a number of performance antipatterns has been detected, instead of blindly evaluating, with a costly performance analysis, all the antipatterns' solutions, a ranking methodology is applied to decide which ones have to be solved. Such ranking process is aimed at estimating the effect of an antipattern solution thus to only focus on the promising ones for conducting a detailed performance analysis and quickly converge towards the desired result of end-users satisfaction.

A first step in this direction was introduced in [11]. Further experimentation conducted in the meanwhile led to arise additional issues not considered before, that we include in the ranking process described in this paper. We realized that the effectiveness of solving antipatterns relies on the joint analysis of multiple performance indices. The effects of solving an antipattern instance for improving a certain performance index (e.g. the response time of a service) are influenced by other performance indices (e.g. the utilisation of processors providing the service), i.e. the factors that likely contribute to understand the consequences of the antipattern refactoring actions.

Compared to our previous work [11, 6], the specific contributions of this paper are: (i) the refinement of the antipattern ranking described in [11]; (ii) the introduction of the refined ranking in the antipattern-based process described in [6]; (iii) the validation of the whole process on four case studies in different application domains.

Ultimately, the benefit of using our approach is that performance analysts can detect and solve performance problems more easily. Instead of manually analysing the resulting indices of performance analyses and coming up with possible alternatives, they only have to apply the refactoring actions generated by our antipattern-based approach.

The paper is organized as follows. Section 2 compares the related work to our approach. An overview of detectable/solvable performance antipatterns within the PCM context is given in Section 3. Section 4 describes in detail some examples of antipatterns that can be detected and solved within the PCM modelling notation: an antipattern is represented as a set of rules for the identification of the problems (see Section 4.1); the detected antipatterns are ranked in order to give a priority to their solution (see Section 4.2); and finally antipatterns are represented as a set of refactoring actions for the application of their solution (see Section 4.3). The advantage of including the ranking activity in the whole antipattern-based process is discussed in Section 5 and explained in Section 6 by means of a leading case study. Section 7 reports the validation of the approach in three different case studies and motivates the beneficial effects of introducing the ranking methodology. Assumptions, limitations, and open issues are discussed in Section 8, and finally Section 9 concludes the paper and gives directions for future research.

2. Related work

In literature few papers deal with the interpretation of performance analysis results and the generation of architectural feedback. Most of them are based on monitoring techniques and therefore are conceived to only act after software deployment for tuning purposes. We are instead interested in model-based approaches that can be applied early in the software lifecycle to support design decisions.

In the following the main existing approaches for the automated generation of architectural feedback are surveyed. In particular, we identified three principal categories: (i) antipattern-based approaches that make use of antipatterns knowledge to cope with performance issues; (ii) rule-based approaches that define a set of rules to overcome performance problems; (iii) search-based approaches that explore the problem space by examining options to deal with performance flaws.

We considered the work done over the last years by Smith and Williams [12] that have defined 14 notation- and domain-independent software performance antipatterns. We are particularly interested in *technology independent* performance antipatterns [12], because our goal is to tackle the problem at the modelling level, by looking at the design of software systems and localizing the most critical parts from a performance perspective.

In [10] we have introduced a technique based on first-order logic to specify system-independent rules that formalize known performance antipatterns. These rules express a set of system properties under which an antipattern occurs with a certain degree of notation-independence. However, for the detection (and consequently the solution) to be applied in practice, we need a software modelling notation that can capture the defined

system properties. In [13] we showed how performance antipatterns can be defined and detected in UML models [14] using OCL queries [15]. However, the solution of antipatterns has not been automated yet, since UML standard profiling (e.g. MARTE [16]) is only aimed at enabling the performance analysis, and it does not allow to specify refactoring actions like the re-deployment of components. In this paper instead the antipattern solution (i.e. the model refactoring) is automatically executed, since it is supported by the PCM Bench tool.

The approach of this paper 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.

Antipattern-based approaches. The term *antipattern* appeared for the first time in [17] in contrast to the trend of focus on positive and constructive solutions. Differently from patterns [18], antipatterns look at the negative features of a software system and describe commonly occurring solutions that generate negative consequences. While architectural and design antipatterns (and patterns) are generally concerned with software quality attributes such as reusability and maintainability [19], performance antipatterns are solely focused on performance concerns.

In [20] the PASA (Performance Assessment of Software Architectures) approach has been introduced. 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 quite low.

In [21] an automated generation of feedback from the software performance analysis driven by antipatterns has been presented, but performance flaws are detected based on the analysis of Layered Queued Network (LQN) models using informal interpretation matrices. Our approach, instead, aims at systematically evaluating performance prediction results by joining the analysis of performance indices (e.g. the utilization of a hardware resource) with the architectural features of software systems (e.g. the interaction among software resources) and, differently from [21], it provides support to automatically solve the detected antipatterns.

The issue of detecting performance antipatterns has been addressed in [22], where a performance diagnosis tool, named Performance Antipattern Detection (PAD), is presented. However PAD only deals with Component Based Enterprise Systems, targeting EJB applications. It is based on monitoring data from running systems, and it extracts the run-time system design and detects EJB antipatterns by applying rules to it. Therefore its scope is restricted to such domain, whereas in our approach the starting point is an architectural model of the software system in the early stages of development.

Rule-based approaches. Such approaches encapsulate the knowledge on how to improve system performance into executable rules that modify the architecture of the system.

In [23] a framework (ArchE) to support the software designers in creating architectures that meet quality requirements has been proposed. It embodies knowledge of quality attributes and the relation between the achievement of quality requirements and design. However, only modifiability tactics are supported.

In [24] the problem of software performance diagnosis and improvement has been tackled, and rules to identify patterns of interaction between resources are defined. Performance flaws are identified before the implementation of the software system, however only the bottlenecks (e.g. the "One-Lane Bridge" antipattern in Smith's classification) and long paths are considered. Additionally, performance issues are identified at the level of a LQN performance model and the translation of these model properties into design changes is a not trivial task due to the gap between software and performance model representations, as outlined in [25]. Differently from [24], our approach refers both to performance and design features of the software system in the feedback generation process in order to keep the information we need to choose the best design alternatives.

Search-based approaches. Such approaches explore the architectural space by examining design options that deal with performance flaws.

In [26] an approach to optimise deployment and configuration decisions has been presented in the context of distributed, realtime, and embedded (DRE) component-based systems. Bin packing algorithms have been enhanced, and schedulability analyses have been used to make fine-grained assignments that indicate how components are allocated 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.

In [27] an approach to automatically explore the design space for hardware architectures has been described. 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 results with a low prediction error. However, the approach is limited to hardware properties, whereas software architectures are more complex, because architectural models spread on a wider range of features (e.g. static and behavioural properties).

In [28] a framework for the optimisation of embedded system architectures has been presented. In particular, it uses the AADL (Architecture Analysis and Description Language) [29] and provides plug-in mechanisms to replace the optimisation engine, the quality evaluation algorithms and the constraints checking. Architectural models are optimised with evolutionary algorithms while considering multiple arbitrary quality criteria. However, the only refactoring action the framework currently supports is the component re-deployment.

In [30, 31], meta-heuristic search techniques are used for improving performance, reliability, and costs of component-based software systems: evolutionary algorithms search the architectural design space for optimal trade-offs. The approach is quite time-consuming, because it mostly uses random changes of the architecture. Some performance knowledge is integrated as performance tactics in [31]. The use of antipatterns can be complementary to meta-heuristic search, because antipatterns can be used as additional tactics to speed up the meta-heuristic search process.

3. Overview of Performance Antipatterns in PCM

In this section we provide an overview of the performance antipatterns that can be specified in the PCM. Section 3.1 first provides basic information on the PCM, then Section 3.2 gives an overview on the detection and solution of antipatterns in the PCM.

3.1. Palladio Component Model Basics

To quickly convey the concepts of the PCM, Figure 2 shows an example of the PCM model elements that are important for antipattern detection and solution, in a simplified UML-like syntax. In the following explanation, the model elements are marked with `Courier font`. Note that only features relevant to this paper are shown here, other PCM features can be found in [7].

A software system in the PCM is modelled as a set of `Basic Components` (e.g. C1, C2 in Figure 2). Components offer `Interfaces`. In the example, `Basic Component` C1 offers `Interface` I1, while `Basic Component` C2 offers `Interface` I2. Additionally, components can require interfaces. In the example, C1 requires the `Interface` I2.

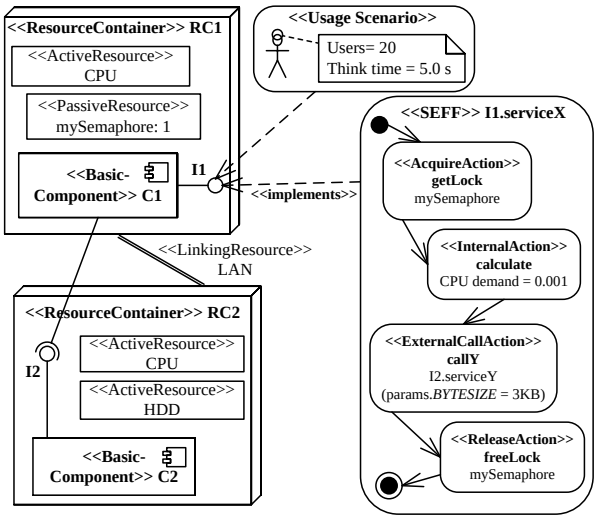

Figure 2: An example of a PCM model.

A PCM model also contains the mapping of software components to hardware, called `Allocation`. Hardware platforms are modelled as `Resource Containers`, which can contain `Active Resources`, such as CPU and hard disk (HDD), or `Passive Resources`, such as semaphores or thread pools. In Figure 2 example, a `Passive Resource` with capacity 1 is modelled in `Resource Container` RC1. `Active Resources` have a `Resource Type` (such as HDD, CPU) and additional properties not shown here, such as a processing rate (how many demand units per second they process) and scheduling policies (such as FCFS or processor sharing). The mapping of components to `Resource Containers` is visualised by placing components inside containers. `Resource Containers` are connected by `Linking Resources`, whose timing behaviour is determined by the amount of sent data.

A `Service Effect Specification` (`SEFF`) describes the behaviour of a service offered by a `Basic Component`. A `SEFF` contains a sequence of actions. An `External Call Action` models calls to required interfaces. For example, serviceX of component C1 calls serviceY of interface I2. As C1 is connected to C2 with this interface, the call is directed to C2's serviceY. Optionally, the size of the passed data can be specified with a `BYTESIZE` characterisation (e.g. 3KB in the example), which is used to determine the linking resource

load. An `Internal Action` specifies a resource demand to an `Active Resource`, such as a CPU or a hard disk (HDD). In the example, serviceX of component C1 has a CPU demand of 0.001 each time it is called. An `Acquire Action` and a `Release Action` model the use of `Passive Resources`.

In the PCM it is possible to specify the `Usage Scenario` that denotes how many users are expected to require a service and their thinking time (i.e. the time a user spends before performing a request). In the example, 20 users with a thinking time of 5 seconds are expected to require the serviceX of the interface I1.

3.2. Performance Antipatterns in the PCM

In general, in a modelling language, there are antipatterns that can be automatically detected and solved, other ones that can be automatically detected, but not automatically solved, and finally other ones that are neither detectable and solvable.

Table 2 lists the performance antipatterns we examine. From the original list of 14 antipatterns defined by Smith and Williams in [12], two antipatterns are not considered for the following reason: the *Falling Dominoes* antipattern refers not only to performance problems, it includes also reliability and fault tolerance issues, and it is out of our interest; the *Unnecessary Processing* antipattern deals with the semantics of processing, by judging the importance of the application code, that is an abstraction level not included in software models.

Table 2 is organized as follows: each row represents a specific antipattern and it is characterized by three fields (one per column), that are: *antipattern* name, and if it is automatically *detectable* and *solvable* in PCM models ($\surd$ yes, – no).

The list of antipatterns has been enriched with an additional attribute: *Single-value* antipatterns are detectable by using only mean, max or min values of performance indices, whereas *Multiple-values* antipatterns must be detected by observing the evolution of the performance indices over time (see more details in [10]).

Table 2 points out that the most interesting antipatterns in the PCM context are: Concurrent Processing Systems, Extensive Processing, and One-Lane Bridge, because they can be automatically detected and solved (see more details in Section 4). Hence, such antipatterns can be referred as *solvable* antipatterns.

Table 2 reveals that there are also five performance antipatterns (i.e. Blob, Pipe and Filter Architectures, Circuitous Treasure Hunt, Empty Semi Trucks, and Traffic Jam) that can be automatically detected, but not automatically solved. Such antipatterns can be referred as *semi-solvable* ones, since it is only possible to devise some actions to be manually performed (see more details in Section 4).

Table 2: Performance Antipatterns in PCM.

	Antipattern		Detectable	Solvable
Single-value	Blob		$\surd$	–
	Unbalanced Processing	Concurrent Processing Systems	$\surd$	$\surd$
		Pipe and Filter Architectures	$\surd$	–
		Extensive Processing	$\surd$	$\surd$
	Circuitous Treasure Hunt		$\surd$	–
	Empty Semi Trucks		$\surd$	–
	Tower of Babel		–	–
	One-Lane Bridge		$\surd$	$\surd$
	Excessive Dynamic Allocation		–	–
Multiple-values	Traffic Jam		$\surd$	–
	The Ramp		–	–
	More is Less		–	–

Finally, Table 2 indicates that there are four performance antipatterns (i.e. Tower of Babel, Excessive Dynamic Allocation, The Ramp, and More is Less) neither detectable nor solvable in the PCM context.

Tower of Babel is an antipattern whose bad practice is on the translation of information into too many exchange formats, i.e. data is parsed and translated into an internal format, but the translation and parsing is excessive [12]. In the PCM, data flow is more abstract and does not include information on data formats. However, it might be possible to replace the current modelling language to specify the behavioural description of services, i.e. the PCM service effect specification (SEFF), with another behavioural description language that includes such information.

Excessive Dynamic Allocation is an antipattern whose bad practice is on unnecessarily creating and destroying objects during the execution of an application [5]. In the PCM, no object-oriented detail is currently available, because it is not included in the current abstraction level. However, it might be possible to detect such bad practice in PCM models that are re-engineered from byte code [32], because constructor invocations are then stored as special type of resource demands at

the modelling layer.

The Ramp is an antipattern whose bad practice is revealed by an increasing value of the response time and a decreasing throughput over time [12]. It might be detected by introducing the concept of *state* as suggested in [33], and it might be possible to inform the architect that a resource demand increasingly grows due to state changes.

More is Less is an antipattern whose bad practice is on the overhead spent by the system in thrashing as compared to the amount of real work, because there are too many processes for the available resources [34]. Thrashing cannot currently be modelled in the PCM, but it might be added by introducing layered execution environment models, as suggested in [35].

4. Antipattern-based process

Figure 3 details the software performance modelling and analysis process of Figure 1. In particular, the refactoring phase is explicitly represented in three main operational steps: (i) *detecting antipatterns* makes use of the problem descriptions of performance antipatterns [12], and localizes problems; (ii) *ranking antipatterns* provides an order in the list of the detected antipatterns by assigning them a score on the basis of requirements violations; (iii) *solving antipatterns* makes use of the solution descriptions of performance antipatterns [12], and suggests the changes to be applied to the software model under analysis.

The refactoring phase can have multiple iterations, as illustrated in Figure 3. This is induced by the stochastic nature of performance antipatterns and, more in general, of performance modelling and analysis. Performance is in fact a quite complex non-functional attribute that is affected by multiple factors. The parameters of models and antipatterns, as well as the performance indices, are typed as stochastic in most of cases (e.g. mean values, probability distribution functions, etc.).

Hence, the detection and solution of one or more antipatterns does not guarantee (by itself) the solution of performance problems in the whole system model. Any model obtained by an iteration of the refactoring phase, where one or more antipatterns have been removed, has to be solved in order to check whether the performance indices of the whole model satisfy the performance requirements. In practice, it has to be checked whether local actions, such as antipattern solutions, sufficiently improve global performance indices.

In some cases antipattern solutions might even degrade the system performance, because the refactoring actions may introduce in the whole model other antipatterns that

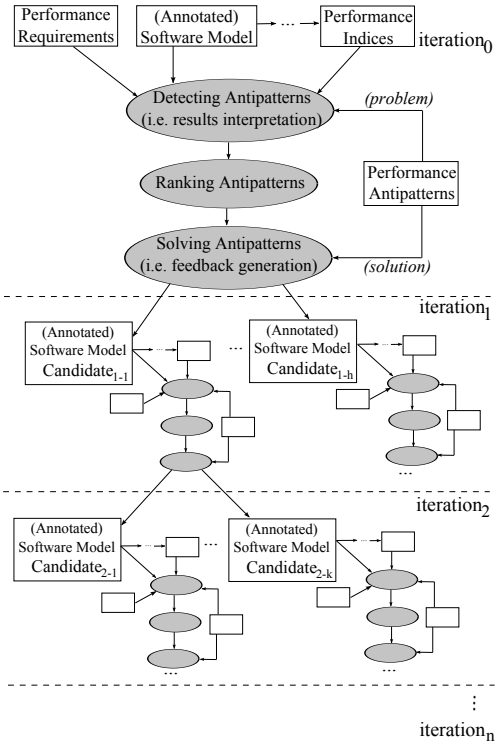

Figure 3: Details of the refactoring phase.

cannot be detected only on the basis of action specifications. For example, an antipattern solution (e.g. the Blob Antipattern) may require to: (i) split a software component in multiple ones to uniformly divide the work; (ii) redeploy the new software components in different hardware platforms to uniformly distribute the load. However it may happen that some of these new components need to communicate among them, thus the network can be exposed to an excessive message traffic (e.g. the Empty Semi Trucks Antipattern).

Hence, if several antipatterns are detected in the initial model, or several refactoring actions are available for a detected antipattern, then the *solving antipatterns* step can produce many refactored software models. Such models come from applying a specific refactoring action and represent a candidate to overcome the model performance problems.

The process can be iteratively applied on each newly generated candidate, leading to a tree of candidate models. In Figure 3, $Candidate_{i-j}$ denotes the j-th software model candidate that is generated in the i-th iteration.

Table 3 lists the performance antipatterns we examine in this section. It reports their natural language definition of the *problem* and *solution* [12]. The list of the antipatterns we consider have been enriched with an

additional attribute: the *semi-solvable* antipatterns can be automatically detected but not automatically solved, whereas the *solvable* antipatterns can be automatically detected and solved (see more details in Section 3).

On the basis of [10], we give the performance antipattern specification in terms of *rules* representing the problem (see Section 4.1) and *actions* representing the solution (see Section 4.3). Both rules and actions refer to PCM meta-model elements.

The ranking methodology is discussed in Section 4.2 and it is aimed at avoiding the explosion of the model candidates number. The basic idea is to specify a priority order for refactoring actions by estimating their effectiveness.

Process afterthoughts are discussed in Section 5 where we propose a more compact graphical representation to summarise the process, and where termination criteria are devised.

4.1. Detecting antipatterns in PCM

Performance antipatterns have been initially defined in textual form, without any underlying formalism, because they were representing the result of practical experiences in performance analysis of real systems [12]. In order to make machine-processable this repository of experience, in [10] we have introduced a formalization of performance antipatterns as a set of rules based on first-order logics.

Of course, similarly to any approach aimed at introducing formalization in an informal domain, these rules resulted from our human interpretation of informal antipattern definitions. Different formalizations could have been raised by different interpretations, however the validation that we have performed in [10] and in the following work gives us a good confidence on the ability of these rules to correctly represent the corresponding performance antipatterns.

On the basis of the formalization in [10], we introduce in this section the rules that allow to detect antipatterns in PCM.

We recall that these rules are aimed at capturing bad practices. So, in order to quantify how bad a practice can be, it is necessary to introduce a set of thresholds representing system features (e.g. the upper bound for the hardware resource utilization). Such thresholds must be instantiated into concrete numerical values, e.g. hardware resources whose utilization is higher than 0.8 can be considered critical ones. The binding of thresholds is not an easy task, but some sources can be used, such as: (i) the system requirements; (ii) the domain expert's knowledge; (iii) the evaluation of the system under analysis. Such binding is out of scope of this paper, however we have initially investigated this issue in [10] and we have recently conducted sensitivity analysis of the antipattern detection task to the threshold values in [36].

Tables 4 and 5 report an excerpt of the thresholds we need to evaluate *software* and *hardware* boundaries. In particular, the first columns of the Tables show the names of the *threshold*s, the second columns provide their *description*, and finally in the third columns *heuristics* are proposed to estimate their numerical values. For example, in Table 4 the $Th_{maxConnects}$ threshold represents the maximum bound for the number of usage relationships a software entity is involved in. It can be estimated as the average number of usage relationships, with reference to the entire set of software instances in the software system, plus the corresponding variance.

Blob is an antipattern whose problem is the excessive message traffic generated by a single class or component [5]. Even though it can be originated by a wrong distribution of either logics or data, the detection of this antipattern can be performed by looking for the effects that are common to both the above cases, i.e. excessive traffic, as formalized in the following PCM rules.

Usage Rule - a complex `Basic Component`, e.g. bc_x, depends on many other basic components, i.e. it requires many `Interfaces` (larger than the $Th_{maxConnects}$ threshold, see Table 4). It might mean that bc_x needs to retrieve a lot of information in order to handle incoming requests.

Interaction Rule - in the behavioural description of a service, i.e. in the `SEFF`, the `Basic Component` bc_x generates excessive message traffic (larger than the $Th_{maxMsgs}$ threshold, see Table 4), i.e. its `External Call Actions` have a high frequency of execution. It might mean that resources managing such communication could suffer from a performance perspective.

Utilisation Rule - if bc_x and the surrounding `Basic Components` (i.e. the basic components with which bc_x communicates) are deployed on the same `Resource Container`, e.g. rc_x, then the performance issues due to the excessive load may come out by evaluating the utilisation of the `ActiveResources` of rc_x (larger than the $Th_{maxHwUtil}$ threshold, see Table 5); otherwise, if basic components are distributed on different `Resource Containers`, then the performance issues due to the excessive message traffic may come out by evaluating the network communication links, i.e. the PCM `Linking Resource` utilisation (larger than the $Th_{maxNetUtil}$ threshold, see Table 5). This rule relays on extracting the utilisation performance index from the simulation results.

Table 3: Examples of performance antipatterns [12].

	Antipattern	Problem	Solution
Semi-Solvable	Blob	Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the application's top-level classes, and to keep related data and behaviour together.
	Circuitous Treasure Hunt	Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look, performance will suffer.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each look).
	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, Session Facade, and Aggregate Entity design patterns provide more efficient interfaces.
Solvable	Concurrent Processing Systems	Occurs when processing cannot make use of available active resources.	Restructure software or change scheduling algorithms to enable concurrent execution.
	Extensive Processing	Occurs when extensive processing in general impedes overall response time.	Move extensive processing so that it does not impede high traffic or more important work.
	One-Lane Bridge	Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn.	To alleviate the congestion, use the Shared Resources Principle to minimize conflicts.

Table 4: Thresholds specification: software characteristics [10].

Threshold	Description	Heuristics
$Th_{maxConnects}$	It represents the maximum bound for the number of usage relationships a software entity is involved	It can be estimated as the average number of usage relationships per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
$Th_{maxMsgs}$	It represents the maximum bound for the number of messages sent by a software entity in a service	It can be estimated as the average number of sent messages per software entity instance, by considering the entire set of software instances in the software system, plus the corresponding variance
...	...	...

Table 5: Thresholds specification: hardware characteristics [10].

Threshold	Description	Heuristics
$Th_{maxHwUtil}$	It represents the maximum bound for the hardware device utilization	It can be estimated as the average number of all hardware devices utilization values, plus the ϵ offset
$Th_{maxNetUtil}$	It represents the maximum bound for the network link utilization	It can be estimated as the average used bandwidth values, with reference to the entire set of network links in the software system, plus the ϵ offset
...	...	...

The output of the detection rules is the set of Basic Components satisfying the defined rules (e.g. bc_x).

Circuitous Treasure Hunt is an antipattern whose problem is an inadequate organization of data that leads the system to look in several places to find the information it needs [12]. The detection of this antipattern in PCM can be performed with the following rules.

DBinteraction Rule - there are at least two Basic Components, e.g. bc_x and bc_y, such that: a) bc_x often calls bc_y, i.e. there is one or more External Call Actions in bc_x's SEFFs that call Interfaces provided by bc_y and that together have a high frequency of execution; and b) bc_y is a database (this is modelled by annotating bc_x with a custom mark).

Utilisation Rule - similarly to the rule with the same name from the Blob antipattern, it is important to check the utilisation of the Active Resources of the Resource Container on which the database basic component bc_y is deployed. Such a Resource Container, e.g. rc_y, is considered critical if the utilisation of any of its Active Resources exceeds a certain threshold.

DiskMoreUtilised Rule - a database access utilizes more hard disks resources than CPU ones. In general, a Resource Container rc_y contains several Active Resources pr_{x_t} of type t, with $t \in \{CPU, HardDisk\}$. This rule matches if the maximum utilisation among all the hard disk(s) in rc_y is higher than the maximum one among all the CPU(s) in rc_y.

The output of the detection rules is a set of Basic Component pairs satisfying the defined rules (e.g. bc_x and bc_y).

Empty Semi Trucks is an antipattern whose problem is an excessive number of requests to perform a task [12]. The detection of this antipattern in PCM can be performed with the following rules.

Interaction Rule - similarly to the Blob antipattern (see the rule with the same name), there is at least one SEFF in which a Basic Component, e.g. bc_x, generates excessive message traffic, i.e. it calls many other services per request. It might mean that resources managing such communication could suffer from a performance perspective.

MessageSize Rule - the Basic Component bc_x sends a high number of messages without optimizing the available bandwidth, i.e. many messages of small size (a small value in the BYTESIZE Characterisation) are exchanged. It might mean that the amount of processing overhead is required many more times than necessary.

RemoteCommunication Rule - the Basic Component bc_x communicates with a high number of remote Basic Components (as captured from the Allocation) that are all deployed on the same remote Resource Container, without optimizing the interface.

The output of the detection rules is the set of Basic Components satisfying the defined rules (e.g. bc_x).

Concurrent Processing Systems is an antipattern whose problem is an unbalanced distribution of workload among the available active resources of a resource type (e.g. CPU) [12]. The detection of the antipattern can be performed with the following rules.

QueueLength Rule - the system cannot make effective use of available Active Resources: there is at least one Active Resource rc_{xt} of Resource Type t in a Resource Container rc_x that has a high average queue length. This rule is evaluated by extracting the queue length performance index of rc_{xt} from the simulation results, and checking if it is higher than a threshold value named $Th_{maxQL}(t)$ (e.g. $Th_{maxQL}(CPU)$ =50 requests and $Th_{maxQL}(HardDisk)$ =70 requests) for that Resource Type t.

Utilisation Rule - the Active Resource rc_{xt} is over utilised. Note that this rule is evaluated by extracting the utilisation performance index of rc_{xt} from the simulation results. The Resource Container rc_x is selected if the utilisation of its Active Resource rc_{xt} exceeds a maximum threshold boundary for its Resource Type t named, for example, $Th_{maxHwUtil}(t)$ (e.g. $Th_{maxHwUtil}(CPU) = 80\%$ and $Th_{maxHwUtil}(HardDisk) = 70\%$).

UnbalancedLoad Rule - active resources are not used in a well-balanced way, namely there is at least another Resource Container instance (e.g. rc_y) whose Active Resources of the same Resource Type t are less utilized in comparison to rc_{xt}. Note that this rule relays on extracting the utilisation performance index of the Active Resources rc_{y_t} from the simulation results. The Resource Container rc_y is selected if the utilisation of rc_{y_t} does not exceed a minimum threshold boundary for that Resource Type t named $Th_{minHwUtil}(t)$ (e.g. $Th_{minHwUtil}(CPU) = 30\%$ and $Th_{minHwUtil}(HardDisk) = 20\%$).

The output of the detection rules is a set of tuples with three elements: two Resource Containers satisfying the defined rules (e.g. rc_x and rc_y) and the Resource Type t (e.g. CPU, HardDisk).

Extensive Processing is an antipattern whose problem is an inefficient management of requests: "lighter" requests are delayed, since they wait for "heavier" ones [34]. The detection of this antipattern in PCM can be

performed with the following rules.

Structural Rule - there are two SEFFs, i.e. $seff_a$ and $seff_b$, that cannot be executed at the same time, due to two different reasons: (i) there is a Branch Action *ba* in a third SEFF, i.e. $seff_c$, which models that either $seff_a$ or $seff_b$ is called from $seff_c$, and *ba* is protected by a Passive Resource *p* of capacity equal to one (i.e. *ba* is preceded by an AcquireAction for *p* and succeeded by a ReleaseAction for *p*); or (ii) there is a FIFO scheduling policy for the Resource Container hosting $seff_a$ and $seff_b$ that disables their concurrency.

ResourceDemand Rule - $seff_a$ has a high global resource demand, i.e. its contained Internal Actions have high resource demands. For example, many CPU units are needed to accomplish a certain task or many bytes are read or written to a hard disk, etc. Such values are compared to threshold values and considered critical whenever they exceed such boundaries.

Probability Rule - Branch Action *ba* in $seff_c$ specifies that the probability of calling $seff_a$ is lower than one, i.e. $seff_a$ must not be always executed.

UnbalancedResDemand Rule - the resource demands for $seff_a$ and $seff_b$ are unbalanced, the former is the heavy one, the latter is the light one, i.e. their resource demands differ of a substantial value.

Utilisation Rule - the Resource Container on which the Basic Component providing $seff_a$ is deployed has a heavy computation, i.e. the utilisation of one of its Active Resources is higher than a threshold value.

The output of the detection rules is the set of SEFF pairs satisfying the defined rules (e.g. $seff_a$ and $seff_b$).

One-Lane Bridge is an antipattern whose problem consists of processes that are not allowed to be processed concurrently [5]. The detection of this antipattern in PCM can be performed with the following rules.

QueueLength Rule - there is at least a Passive Resource, e.g. pr_x, that has a large queue length, i.e. its queue length is higher than a threshold value.

WaitingTime Rule - the requests incoming to the Passive Resource pr_x are delayed, i.e. the time they hold pr_x is much smaller than the time they have to wait for pr_x. Note that this rule is evaluated by extracting the holding time and the waiting time performance indices of pr_x from the simulation results.

The output of the detection rules is the set of passive resources satisfying the defined rules (e.g. pr_x).

Based on these rules, antipattern instances are detected by a search algorithm as follows. For each antipattern, all instances of the main metamodel element are retrieved in the PCM software model. The main metamodel element of an antipattern is the first metamodel element mentioned by the antipattern. For example, for the Blob antipattern all Basic Components are retrieved and for the Concurrent Processing Systems antipattern all Active Resources are retrieved. An exception is the Extensive Processing antipattern, which has two main metamodel elements, Passive Resource (for the case that a protected branch causes the extensive processing) and Resource Container (for the case that the resource container's scheduling causes the extensive processing). In this case, all instances of both main metamodel elements are retrieved.

Then, for each retrieved model element instance, the rules are checked. If any rule evaluates to false, the model instance is dismissed without evaluating possibly remaining rules (short-circuit evaluation). If all rules match, the model element instance is returned together with additional information specified for the antipattern.

4.2. Ranking Antipatterns

Figure 4 details the software performance modelling and analysis process of Figure 3. In particular, shaded boxes of Figure 4 represent the *ranking antipatterns* operational activity that is object of this section.

We discuss the problem of identifying, among a set of detected performance antipatterns, the guilty ones, i.e. the antipatterns that are the actual causes of requirements violations. A process to elaborate the performance analysis results and to score performance antipatterns on the basis of violated requirements is introduced. The cross observation of such scores allows to classify the level of effectiveness of each antipattern.

We recall that a first step in this direction has been presented in [11], but in the meanwhile more experience has been collected and further issues are considered in the following.

Performance requirements are classified on the basis of the performance indices they address and the level of abstraction they apply. Our experience leads us to focus on the most common types of requirements that concern: *utilisation* of active resources, *response time* and/or *throughput* of basic and composed services. "Basic service" denotes a functionality that is provided by a component without calling services of other components, and "composed service" denotes a functionality that is provided by a component and involves a combination of calls to services of other components.

The *Violated requirements* (see Figure 4) are related to the performance indices that are not satisfied. Each requirement is represented by: (i) an identifier (*ID*), (ii) the type of requirement (*Requirement*) that summarizes

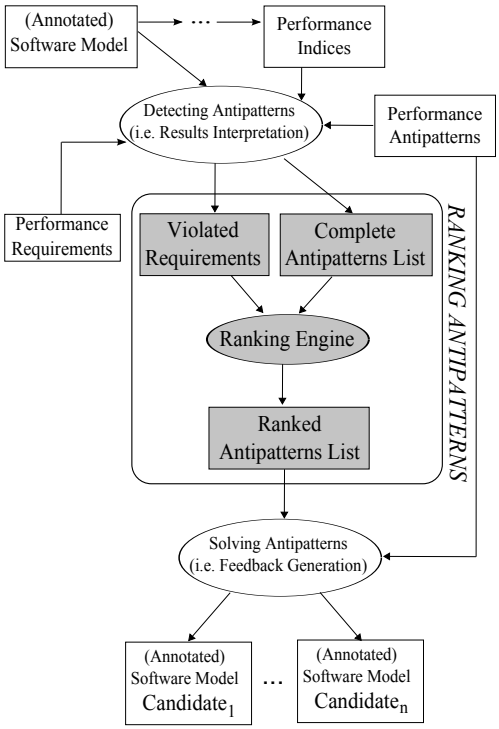

Figure 4: A step ahead in the antipatterns solution.

the performance index and the target software model element, (iii) the required value of the index (*Required Value*), (iv) the observed value as obtained from the performance analysis (*Observed Value*). Based on this information, we can derive (v) the software model entities involved in the requirement (*Involved Entities*) from the software model.

In Table 6 three examples of violated performance requirements are reported. The first one refers to the utilisation index (i.e., U): it requires that the `Resource Container` rc_1 is not utilised more than 70% while it shows an observed utilisation of 74%. The second one refers to the response time index (i.e., RT): it requires that the `SEFF` $seff_1$ has a response time not larger than 2 seconds, whereas it shows an observed response time of 3.7 seconds. The third one refers to the throughput index (i.e., T): it requires that the `SEFF` $seff_4$ has a throughput higher than 5 requests/second, whereas it shows an observed throughput of 1.5 requests/second.

For violated requirements, the involved entities can be derived from the software model as follows. For utilisation requirements, we only consider as involved the `Resource Container` for which the requirement is specified. For example, if a utilisation requirement has been specified for the `Resource Container` rc_1, we consider only rc_1 to be involved. For requirements on `SEFF`s (e.g. response time, throughput), all the `SEFF`s that participate in the service provisioning are considered as involved. For example, if a violated requirement is specified for a `SEFF` $seff_1$, and $seff_1$ itself calls `SEFF`s $seff_2$ and $seff_3$, we consider all these `SEFF`s to be part of the requirement. All the `Basic Components` that provide $seff_1$, $seff_2$ and $seff_3$ are considered as involved entities (e.g. bc_1, bc_2 and bc_3), as well as all the `Resource Containers` hosting them (e.g. bc_1 is deployed on rc_1, bc_2 is deployed on rc_2, and bc_3 is deployed on rc_3). The goal is to capture the model entities that most likely cause the observed performance problems.

The *Complete antipatterns list* (see Figure 4) groups all the detected antipatterns. An example is reported in Table 7, where each antipattern is denoted by: (i) an identifier (*ID*), (ii) the type of antipattern (*Detected Antipattern*) as specified in the Smith-Williams classification [12], (iii) the software model entities involved in it (*Involved Entities*), (iv) the performance indices that do not likely contribute to the antipattern occurrence (*Contributing Indices*).

In Table 7 three examples of detected antipatterns are reported. The first one refers to the Concurrent Processing Systems antipattern (i.e., CPS): the output of the detection (see Section 4.1) is constituted by two `Resource Containers` satisfying the defined rules (e.g. rc_x and rc_y) and the critical resource type t (e.g. CPU, HardDisk). We insert in the involved entities column only rc_y because it represents the least used active resource and will be used in the solution of the antipattern. The performance indices that likely contribute to the antipattern occurrence are the utilisations of both the `Resource Containers`. The second example of Table 7 refers to the Extensive Processing antipattern (i.e., EP): the output of the detection is constituted by the `SEFF` pair satisfying the defined rules (e.g. $bc_a.seff_a$ and $bc_b.seff_b$). The performance indices that likely contribute to the antipattern occurrence are the throughputs of both `SEFF`s. The third example of Table 7 refers to the One-Lane Bridge antipattern (i.e., OLB): the output of the detection is constituted by the `Passive Resource` satisfying the defined rules (e.g. pr_x). We insert in the involved entities column the `Basic Components` (e.g. bc_1, bc_2) using pr_x. The performance indices that likely contribute to the antipattern occurrence are the utilisations of the `Passive Resource` and the `Resource Container` (e.g. rc_1) on which it is deployed on.

The *ranking engine* (see Figure 4) is aimed at ordering the list of the detected antipatterns for each violated requirement, where highly ranked antipatterns are the most promising causes for the requirement violation.

Table 6: Example of *Violated Requirements*.

ID	Requirement	Required Value	Observed Value	Involved Entities
R_1	U(rc_1)	0.70	0.74	rc_1
R_2	RT($seff_1$)	2 sec	3.7 sec	$bc_1.seff_1, bc_2.seff_2, bc_3.seff_3, rc_1, rc_2, rc_3$
R_3	T($seff_4$)	5 $\frac{reqs}{sec}$	1.5 $\frac{reqs}{sec}$	$bc_4.seff_4, rc_4$
...	...	...	...	...

Table 7: Example of *Complete Antipatterns List*.

ID	Detected Antipattern	Involved Entities	Contributing Indices
PA_1	CPS	rc_y	U(rc_x), U(rc_y)
PA_2	EP	$bc_a.seff_a$, $bc_b.seff_b$	T($seff_a$), T($seff_b$)
PA_3	OLB	bc_1, bc_2	U(pr_x), U(rc_1)
...	...	...	...

In this context, a performance antipattern PA_i is considered a cause for the violation of the requirement R_j if the intersection set of involved model entities $\{e_1, .., e_k\}$ is not empty, namely if the *Involved Entities* columns of Tables 6 and 7 have common entities. Antipatterns that do not have any entity in common with a violated requirement will be considered unaffecting for the latter.

Our ranking process starts by considering the model entities involved in a violated requirement: a score is assigned to each entity, and all the involved entities will contribute to the final rank of the antipattern.

In [11] we introduced a set of equations to rank model entities related to utilisation, throughput, and response time requirements. Table 8 reports the equations that we introduced in [11] to assign scores to system entities involved in utilisation and throughput violated requirements. From more recent experiences we deduced that the violation of the response time requirement can be refined, as reported in Table 9 and explained in the following after briefly sketching the previous scores.

Utilisation. The violation of an utilisation requirement can only target (in this paper scope) a processor. For each violated requirement R_j, we introduce a utilisation score to the involved processor $Proc_i$ as reported in the first row of Table 8. $score_{i,j}$ represents a value between 0 and 1 that indicates how much the $Proc_i$ observed utilisation ($observedUtil_i$) is higher than the required one ($requiredUtil_j$).

Throughput. The violation of the throughput in composed services involves all services participating to the end-user functionality. For each violated requirement R_j, we introduce a throughput score to each involved service S_i as reported in the third row of Table 8. We distinguish between open and closed workloads here. For an open workload ($isOpen(systemWorkload)$), we can identify bottleneck services S_i that cannot cope with their arriving jobs ($workload_i > observedThrp_i$). To these services a positive score is assigned, whereas all other services are estimated as not guilty for this requirement violation and a score of 0 is assigned to them. For closed workloads ($isClosed(systemWorkload)$), we always observe job flow balance at the steady-state and thus for all services $workload_i = observedThrp_i$ holds. Thus, we cannot easily detect the bottleneck service and we assign a positive score to all involved services. For the positive scores, we quantify how much the observed throughput of the overall composed service ($observedThrp_j$) is far from the required one ($requiredThrp_j$). The violation of the throughput in basic services involves just this one service. We can use the previous equation as it is, because the only involved service is the one under stress.

Response time. In [11] the response time requirement was only assigning scores to the services S_i, as reported in the first row of Table 9. From more recent experiences we deduced that the violation of the response time in composed services CS_j involves not only all services participating in that functionality (denoted by $S \in CS_j$), but also active resources play an important role. For this reason we introduce in this paper a response time score also for the active resources P_k for each violated requirement R_j, as reported in the second row of Table 9. We quantify how far the observed mean response time of the composed service CS_j ($observedRespTime_{CS_j}$) is from the required one ($requiredRespTime_{CS_j}$). Additionally, in order to increase the guilt of basic services and active resources that mostly contribute to the response time of the composed service, we introduce the first multiplicative factor of the equation.

We denote with $ownComputation_{S_i}$ the observed computation time of a service S_i participating in the composed service CS_j. If service S_i is a basic service, $ownComputation_{S_i}$ equals the mean response time $RT(S_i)$ of service S_i. However, composite services can also consist of other composite services. Thus, if service S_i is a composite service that calls services S_1 to S_n

Table 8: Ranking of performance antipatterns for the utilisation and throughput requirements [11].

Requirement	Equation
Utilisation	$score_{i,j} = (observedUtil_i - requiredUtil_j)$
Throughput	$score_{i,j} = \begin{cases} \frac{requiredThrp_j - observedThrp_j}{requiredThrp_j} & \text{if } workload_i > observedThrp_i \text{ or } isClosed(systemWorkload) \\ 0 & \text{else} \end{cases}$

Table 9: Ranking of performance antipatterns for the response time requirement.

Requirement	Equation
Response time	$score_{S_i,CS_j} = \frac{ownComputations_{S_i}}{maxOwnComputationS_{CS_j}} \cdot \frac{observedRespTime_{CS_j} - requiredRespTime_{CS_j}}{observedRespTime_{CS_j}}$ for $S_i \in CS_j$
	$score_{P_k,CS_j} = \frac{ownComputation_{P_k}}{maxOwnComputationP_{CS_j}} \cdot \frac{observedRespTime_{CS_j} - requiredRespTime_{CS_j}}{observedRespTime_{CS_j}}$

with frequencies $F(S_1)$[3] to $F(S_n)$, $ownComputation_{S_i}$ is defined as the total mean response time of service S_i minus the weighted mean response time of called services:

$$ownComputation_{S_i} = RT(S_i) - \sum_{1 \le c \le n} F(S_c) RT(S_c)$$

We divide by the maximum own computation over all the services participating in CS_j, which we denote by $maxOwnComputationS_{CS_j}$:

$$maxOwnComputationS_{CS_j} = \max_{S_i \in CS_j} (ownComputation_{S_i})$$

In this way, services with higher response time will be more likely retained responsible for the requirement violation.

We denote with $ownComputation_{P_k}$ the observed computation time of an active resource P_k participating in the composed service CS_j. It is calculated by summing the own computation(s) of service(s) whose basic components are deployed on the active resource P_k:

$$ownComputation_{P_k} = \sum_{1 \le z \le m} ownComputation(S_z)$$

Let PR_{CS_j} denote the set of all active resources participating in the composed service CS_j. We divide by the maximum own computation over all the active resources participating in CS_j, which we denote by $maxOwnComputationP_{CS_j}$:

$$maxOwnComputationP_{CS_j} = \max_{P_k \in PR_{CS_j}} (ownComputation_{P_k})$$

In this way, active resources with higher computation requests will be more likely retained responsible for the requirement violation.

Ranking refinement. The ranking described above is based on the involvement of software model entities in detected antipatterns and in violated requirements. The defined scores help to order the detected antipatterns on the basis of their guilt. However, after having experienced our approach on several examples, we have realized that the feasible refactoring actions available for a highly ranked antipattern do not necessarily bring to the best performance improvements, due to limitations in the antipattern context.

Therefore, we introduce here the concept of *semantic factor* that aims at capturing the potential effectiveness of an antipattern solution by taking into account additional performance indices that are not directly involved in the antipattern definition. A semantic factor is introduced to capture the semantic nature of a performance antipattern, and it represents a quantification of a property that is antipattern-specific. Hence, each antipattern may have its own semantic factor. For example, when the solution of an antipattern suggests the redeployment of one component from an over-utilised device to an under-utilised one, such refactoring action is as more effective as larger is the gap between the utilization of the two devices. Hence, this gap must enter the guilt degree definition for such an antipattern. This would allow to identify the cases where it is preferrable to act on lower ranked antipatterns that provide refactoring actions with higher potential in terms of performance improvement. Hence, we have defined several semantic factors that,

[3] We denote by $F(S_1)$ how often the service S_1 is invoked within the execution of the composite service S_i on average.

once opportunely combined with the scores introduced in the previous section, ends up with refined guilt scores that can more quickly drive the refactoring phase towards the end, as it will be shown in Section 7.

In table 10 a *semantic factor* is introduced for each antipattern type (i.e. sf_{PA}).

Table 10: Semantic factors for performance antipatterns.

Antipattern	Semantic Factor
CPS	$sf_{CPS} = \lvert U(rc_x) - U(rc_y)\rvert$
EP	$sf_{EP} = \frac{\lvert T(seff_a) - T(seff_b)\rvert}{max\{T(seff_a), T(seff_b)\}}$
OLB	$sf_{OLB} = \lvert U(rc_x) - U(pr_x)\rvert$
...	...

The semantic factor in the first row refers to the Concurrent Processing Systems antipattern (i.e., CPS): it is defined as the gap among the utilisations of active resources (output of the detection rules), since this antipattern solution aims at moving some computation from the most utilised active resource toward the less utilised one. The second row refers to the Extensive Processing antipattern (i.e., EP): its semantic factor is calculated as the gap among the throughput of services (output of the detection rules), since the antipattern solution is aimed at improving the scheduling of such services. The third row refers to the One-Lane Bridge antipattern (i.e., OLB): its semantic factor is calculated as the gap among the utilisations of the passive resource (output of the detection rules) and the resource container hosting such resource, since the antipattern solution is aimed at improving the concurrency on processing requests.

Combining the scores of entities. Finally, we rank the antipatterns involved for each violated requirement R_j. A guilt degree $GD_{PA_x}(R_j)$ that measures the effectiveness of PA_x for R_j is assigned to each antipattern PA_x that shares involved entities with a requirement R_j. We define the guilt degree as the sum of the scores of all the involved entities and its corresponding semantic factor:

$$GD_{PA_x}(R_j) = \sum_{i \in involvedIn(PA_x, R_j)} score_{i,j} + sf_{PA_x}$$

The score of each involved entity varies in the interval $[0, \ldots, 1]$ as well as the semantic factor. Hence, the guilt degree of each antipattern varies in the interval $[0, \ldots, n+1]$, where n is the number of involved entities in the violated requirement.

The *ranked antipatterns list* (see Figure 4) adds guilt degrees to all detected antipatterns. Table 11 groups

Table 11: Structure of *Ranked Antipatterns List*.

Requirements	Detected Antipatterns		
	...	PA_x	...
...			
R_j	...	$GD_{PA_x}(R_j)$	...
...			

antipatterns and requirements. A concrete example of a ranked antipatterns list has been reported in the case study section (see Table 15).

Different types of analysis can originate from a ranked list, as it can be analysed by columns or by rows. Firstly, by rows, we concentrate on a certain requirement, for example R_j, and we look at the scores of antipatterns. Antipatterns that are more guilty for that requirement violation can thus be identified. Observing the table by columns, instead, we can distinguish either the antipatterns that most frequently appear in the violation of requirements or the ones that sum up to the highest total degree of guilt.

The solution step of the antipattern-based process in Figure 4 takes as input the ranked antipatterns list, and it makes use of ranking estimates to focus on a set of candidates and to discard some others.

4.3. Solving antipatterns in PCM

Similarly to the antipattern problem definitions, even antipattern solution actions have been only informally defined in literature [5]. However, the formalization of antipattern problems in first-order logics that we have introduced in [10] has helped us to more precisely define refactoring actions that can solve an antipattern.
Let us remark that, due to the typical disjoint formulation of antipatterns (i.e. they can be usually expressed as an AND of predicates), oftenly the available solution is not unique, because different refactoring actions could bring to negate one of the predicates in the antipattern formulation [10]. We will discuss in Section 5, among other, the issues related to the automation of antipattern solution and, more in general, of the whole process.
In this section we describe the refactoring actions that can be applied to solve different antipatterns in PCM.

Note, however, that Blob, Circuitous Treasure Hunt, and Empty Semi Trucks are antipatterns whose solution cannot be automated in PCM because components are considered black-box elements, and the component's internal behaviour cannot be restructured. Alternatively, some actions can be suggested to the architects, and they can manually specify the alternative component(s) able to substitute the detected one(s).

Blob is an antipattern whose solution can be performed by delegating the business logics of the Blob basic component to the ones with which it communicates, e.g. by decreasing the number of required interfaces and/or the number of calls.

Circuitous Treasure Hunt is an antipattern whose solution can be performed with two actions. The first action is aimed at decreasing database communications by restructuring the component interacting with the database to avoid excessive communication. The second action aims at refactoring the database component in its internal structure by organizing it in such a way that database requests can be performed without accessing too many tables.

Empty Semi Trucks is an antipattern whose solution can be performed with three actions. The first action is aimed at avoiding excessive remote communication by redeploying the component responsible for it, thus to not overload network resources. The second action is aimed at optimizing the usage of the bandwidth by reducing the number of sent messages; it can be performed by batching the messages, i.e. collecting small messages in fewer messages of larger sizes. The third action is aimed at optimizing the usage of the interface by reducing the remote communication; it can be performed by delegating the requests of a component to another one that is remotely deployed with the other communicating components.

Concurrent Processing Systems, Extensive Processing, and One-Lane Bridge are antipatterns whose solution can be automated in the PCM by devising a set of refactoring actions explained in the following.

Concurrent Processing Systems is an antipattern whose solution looks at restructuring software or changing scheduling algorithms [12]. The solution of the antipattern can be performed with one of the following actions.

BalanceLoad Action - if some of the `Basic Components` on the `Resource Containers` rc_x and rc_y[4] offer the same `Interfaces`, change the scheduling algorithms and distribute the requests for such services in a balanced way (from rc_x to rc_y) by modifying the probability to be called.

Mirror Action - mirror the `Basic Components` of the `Resource Container` rc_x into rc_y and balance the workload, so that the requests incoming to the system are distributed to both `Resource Containers`. Consider the available `Active resources` (i.e. cpu(s), hard disk(s)) of the `Resource Containers`.

MostCritical Action - identify the `Basic Component` of the `Resource Container` rc_x that has the highest resource demand of the critical type t, and redeploy it in the `Resource Container` rc_y.

Redeploy Action - redeploy some `Basic Components` from the `Resource Container` rc_x to rc_y. Such action can be performed by taking into account a set of system properties or their combination, as argued in the following.

One option is to equally distribute the resource demand of type t among `Resource Containers` rc_x and rc_y by redeploying some `Basic Components` from rc_x to rc_y. We select those components for which the `Resource Container` rc_y has sufficient resource capacity for the other resource type (i.e. CPU or HDD).

A second option is to redeploy components on the basis of their communication and trying to deploy components communicating with each other, possibly on the same `Resource Container`.

Extensive Processing is an antipattern whose solution looks at scheduling requests according to their processing load and/or relevance [34]. The solution of the antipattern can be performed with one of the following actions, depending on which *structural rule* of the antipattern was satisfied (see Section 4.1).

IncreaseCapacity Action - increase the capacity of the `Passive Resource` that does not allow concurrency while executing $seff_a$ or $seff_b$ in the `Branch Action` (if case (i) of the structural detection rule was matched).

UnblockExecution Action - change the scheduling algorithm of the resource and/or redeploy one of the `Basic Components` containing $seff_a$ or $seff_b$ so they do not queue for the same `Active Resource` anymore (if case (ii) of the structural detection rule matched).

One-Lane Bridge is an antipattern whose solution looks at sharing resources thus to avoid congestion of requests [5]. The solution of the antipattern can be performed with the following action.

IncreaseCapacity Action - increase the capacity of the `Passive Resources` by one. A smarter methodology can be devised to optimize the capacity by estimating the minimal multiplicity able to solve performance issues.

5. A deeper look at the whole process

In this section we discuss several issues related to the whole antipattern-based process, with the aim of clari-

[4]Note that rc_x and rc_y represent the instances coming from the antipattern detection (see Section 4.1).

fying specific aspects of our approach.

Figure 5 depicts the process we presented in Figure 3 in a graph-like way: each node represents a (annotated) software model and its performance indices; each arc represents a refactoring *action* applied to solve a detected *antipattern*.

Each node additionally stores the *requirement*s under analysis and their *observed value*s, since such requirements represent what end-users expect from the system for the target performance properties to be fulfilled. Such graphical representation gives an immediate overview on the software model(s) that might best fit the end-users requirements.

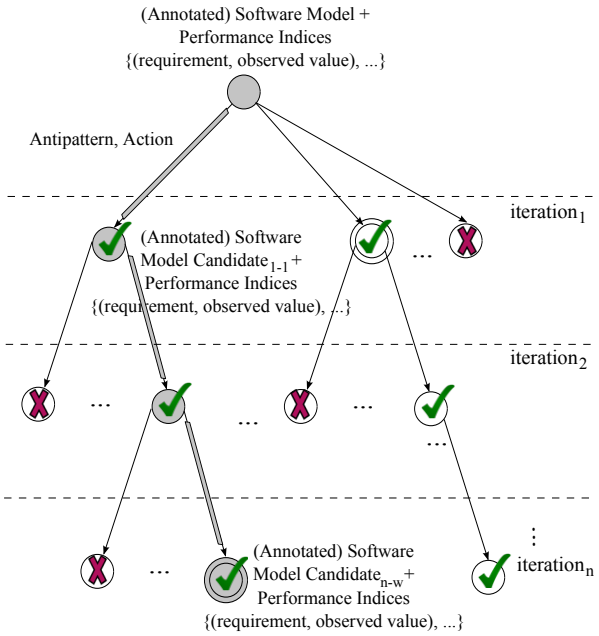

Figure 5: Graph-like representation of our process.

From a given software model, multiple design alternatives may be suggested in the light of detected antipatterns. These alternatives can be originated not only by the decision of solving different antipatterns, but also by different refactoring actions that may be available to remove the same antipattern, as mentioned in Section 4.3. Each option results in a child software model, thus eventually generating a *tree* of candidates as the antipatterns-based approach progresses[5].

[5]Note that we decided to separately consider each antipattern action (i.e. a single action gives rise to a software model candidate), because antipattern actions may overlap each other, e.g. the redeployment of the same component to different `Resource Container`s. Hence, we leave the study of contemporary refactoring actions to further investigation.

The whole process has been implemented as an extension to the PCM Bench tool[6]. The current implementation can solve three antipatterns in PCM models, namely Concurrent Processing Systems, Extensive processing and One-lane Bridge. They in fact represent, as illustrated in Section 3.2, the antipatterns whose solution can be automated within the PCM syntax boundaries. Semi-solvable antipatterns are only detected in PCM, but they would need a human contribution to be solved by hand.
This is not true in general, because different modeling notations could offer different potential to the detection and solution steps, like we have shown in UML [13] and in the Æmilia ADL [37].

The tool completely automates the detection, the ranking, and the solution steps. Therefore, the role of architects in this context is limited to the design of a (annotated) software model and to the specification of performance requirement(s). PCM Bench returns the whole tree illustrated in Figure 5, where the best ranked options have been processed.

As mentioned in Section 4, the solution of one or more antipatterns does not guarantee performance improvements in advance, because the entire process is based on heuristic evaluations and acts within a stochastic domain. Therefore, different termination criteria can be defined for the process: (i) *fulfilment* criterion, i.e. all requirements are satisfied and a software model able to cope with user needs is found; (ii) *no-actions* criterion, i.e. no antipatterns are detected in the software models therefore no refactoring actions can be further experimented; (iii) *#iterations* criterion, i.e. the process can be terminated if a certain number of iterations have been completed.

Full automation of this process could not be the best scenario in several cases. Human experience can be useful, between one iteration and the next one, to cut certain alternatives or to assign high priority to other alternatives, so to drive the whole process. These choices can be based on domain-specific constraints, budget limits, legacy constraints, etc. that are hard to codify within the process to be machine-processable. However, in this paper we aim at describing and validating the correctness and effectiveness of our approach to solve performance problems, whereas we leave user interaction aspects for further investigation.

The representation we propose in Figure 5 allows to associate the software performance feedback to a path in the tree of candidates: all the actions specified in the

[6]PCM Bench and antipattern-based extension can be downloaded at sdqweb.ipd.kit.edu/wiki/PerOpteryx.

path indicate the design alternatives to be assessed in the software model. It may happen that the best candidate of one iteration, if any, i.e. the local optimum (represented with a double circle), is not necessary included in the path towards the final best candidate (depicted with shaded arcs), i.e. the global optimum.

The goal of the ranking process is to support the antipatterns' solution step in order to quickly converge towards the desired performance improvement, but without compromising the path towards the final best candidate. On the basis of the ranked antipatterns list, different *selection criteria* can be devised to decide how to proceed in the solution step: (i) *limit* criterion, i.e. all antipatterns whose guilt degree is lower than a limit value (e.g. 0.5) are discarded, or (ii) *percentage* criterion, i.e. a percentage of the detected antipatterns (e.g. 40%) will be discarded, that are the ones with the lowest guilt degrees. We have used both the above criteria in our case studies, as it will be illustrated case-by-case.

Once a selection criterion has been defined, the tree of software model candidates can be pruned. In fact in Figure 5 we distinguish the nodes with the following meaning: $\surd$, i.e. the corresponding antipattern action is experimented; $\times$, i.e. the corresponding antipattern action is discarded. In this way our guilt-based strategy operates like a *branch and bound* algorithm that discards subsets of fruitless candidates.

6. A leading case study

In this Section we discuss an illustrative case study to demonstrate the validity of the antipattern-based process, and it is organised as follows. First, Section 6.1 describes the PCM model of the system under analysis, the so-called business reporting system. Then, the stepwise application of the antipattern-based process is performed. The first iteration is described in Section 6.2, whereas Section 6.3 presents the application of the whole process across multiple iterations.

The experimentation is conducted as follows. Starting from the system modelled in PCM, for performance analysis the simulation code has been generated and executed with SimuCom [7]. The simulation results are interpreted by our tool and may reveal performance issues in the system if the prediction value of one or more performance indices does not fulfil the requirements.

Then, the antipattern-based process is applied: if some performance antipatterns are detected in the model, they are ranked and their solution suggests the architectural alternatives that lead to obtain new software model candidates. In this case study we adopt the ranking methodology with the *percentage* selection criterion (see Section 5), hence we discard 40% of the detected antipatterns with the lowest guilt degrees. The software model candidates (that are not discarded by our ranking methodology) are iteratively analysed with the same process until a candidate able to satisfy the performance requirements under study is found, or until no antipatterns are detected any more.

We demonstrate that the introduction of the ranking methodology in the solution step leads the system to converge towards the desired performance improvement by discarding 34% of design alternatives.

For sake of simplification, our experimentation is focused on the analysis of the response time of the system (i.e. the average time a user spends in the system), and it must not overcome 27.5 seconds, under an expected average workload of 30 requests per second with thinking time of 5 seconds.

6.1. Business Reporting System

The system under study is the so-called Business Reporting System (BRS), which lets users retrieve reports and statistical data about running business processes.

Figure 6 shows an overview of the PCM software model for the BRS system. It is a 4-tier system consisting of several basic components, as described in the following. The *Webserver* handles user requests for generating reports or viewing the plain data logged by the system. It delegates the requests to a *Scheduler*, which in turn forwards the requests. User management functionalities (e.g. login, logout) are directed to the *UserManagement*, whereas report and view requests are forwarded to the *OnlineReporting* or *GraphicalReporting*, depending on the type of request. Both components use a *CoreReportingEngine* for the common report generation functionality. The latter one frequently accesses the *Database*, but for some request types uses an intermediate *Cache*. The allocation of software components on resource containers is shown in Figure 6, e.g. $Proc_2$ deals with the scheduling of requests by hosting Scheduler, UserManagement, OnlineReporting and GraphicalReporting basic components.

The system supports seven use cases: users can login, logout and request both reports or views, each of which can be both graphical or online; administrators can invoke the maintenance service.

Not all services are inserted in Figure 6 for sake of readability, however two examples are shown: *SEFF onlineReport* of component *OnlineReporting* implements the interface *IOnlineReporting*, and *SEFF graphicalReport* of component *GraphicalReporting* implements the interface *IGraphicalReporting*. Both services

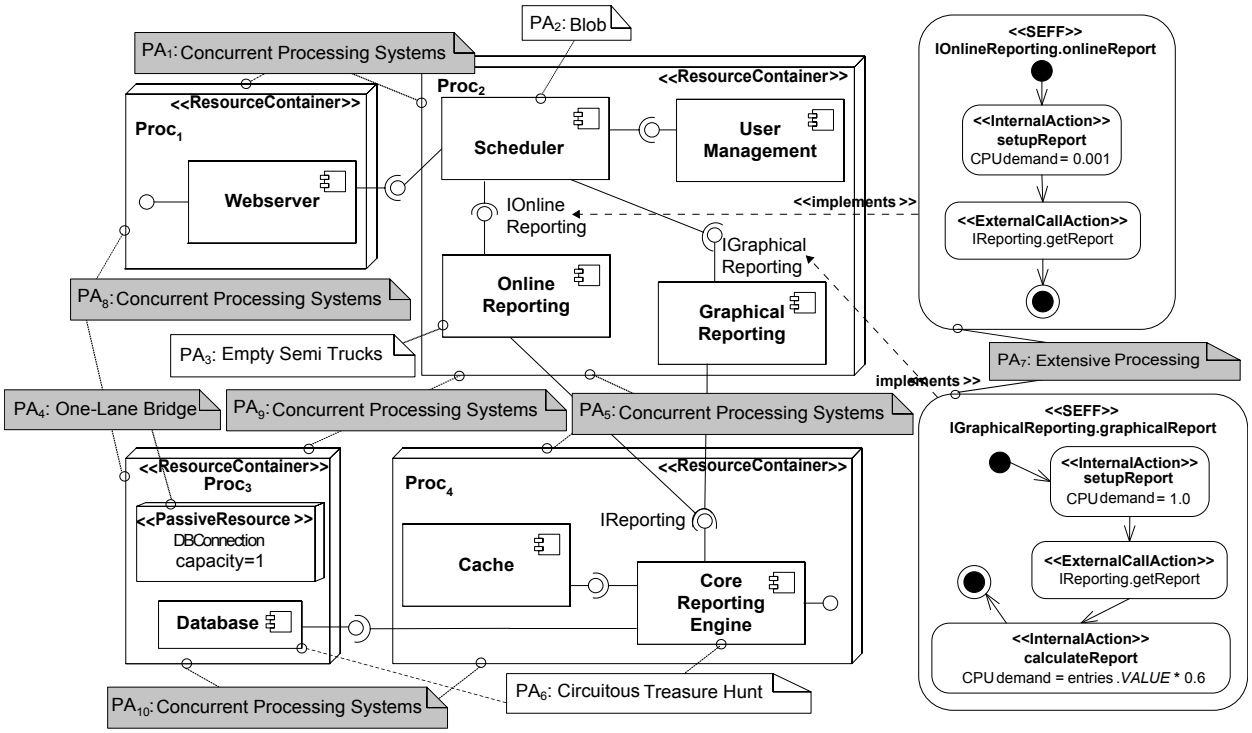

Figure 6: PCM Software Model for the BRS system.

have an *InternalAction* to setup the report and then an *ExternalCallAction* demands to get the report from the CoreReportingEngine component. For the graphicalReport service is necessary to additionally calculate the report for each requested entry. Each internal action is annotated with a resource *demand* indicating the time spent for processing such operation, e.g. the setup of the onlineReport requires 0.001 CPU units.

The PCM software model contains the static structure, the behaviour specification of each component and it is annotated with resource demands and resource environment specifications. Additionally, the PCM software model contains the usage model specifying how users use the system: users login, 25 times the onlineView service is invoked, 5 times the graphicalView and onlineReport services are invoked, and finally the graphicalReport and maintain services are performed before the logout.

For performance analysis, the software model is automatically transformed to simulation code, which is executed by the SimuCom simulation [7]. The performance analysis of the BRS software model reveals that the response time of the system is 46.34 seconds (under the expected workload of 30 requests per second with thinking time of 5 seconds), so it does not meet the stated requirement, i.e. 27.5 seconds. Since the requirement is not satisfied, we apply our approach to detect, rank and solve performance antipatterns.

6.2. First iteration of the antipattern-based process

In this section we discuss the first iteration of the antipattern-based process, in particular we describe the detection of antipatterns (see Section 6.2.1), the ranking (see Section 6.2.2) and their solution (see Section 6.2.3).

6.2.1. Detecting Antipatterns

The labels in Figure 6 indicate the detected antipatterns. Ten instances of antipatterns are found (PA_1, ..., PA_{10}), e.g. the Concurrent Processing Systems for $Proc_1$ and $Proc_2$, the Blob is recognized in the Scheduler component, the Empty Semi Trucks is associated to the OnlineReporting component, and so on. Shaded labels represent the solvable antipatterns, i.e. the ones that we consider for the ranking and the solution.

Table 12 reports some examples of the detected antipatterns for the BRS software model in the first iteration. The first column contains the *antipattern* type according to the Smith and Williams classification [12];

Table 12: BRS- examples of detected antipatterns.

Antipattern	Problem
PA_1 - Concurrent Processing Systems	*QueueLength Rule* - the PCM Active Resource CPU of $Proc_2$, not shown in Figure 6 for sake of readability, has a queueLength of 2.2 requests (i.e. greater than the threshold value, $Th_{maxQL}(CPU)$ =1.5 requests); *Utilisation Rule* - the PCM Active Resource CPU of $Proc_2$ has an utilisation of 49% (i.e. greater than the threshold value, $Th_{maxHwUtil}(CPU)$ =40%); *UnbalancedLoad Rule* - the PCM Active Resource CPU of $Proc_1$ has an utilisation of 0.5% (i.e. lower than the threshold value, $Th_{minHwUtil}(CPU)$ =10%).
PA_5 - Concurrent Processing Systems	*QueueLength Rule* - the PCM Active Resource CPU of $Proc_2$ has a queueLength of 2.2 requests (i.e. greater than the threshold value, $Th_{maxQL}(CPU)$ =1.5 requests); *Utilisation Rule* - the PCM Active Resource CPU of $Proc_2$ has an utilisation of 49% (i.e. greater than the threshold value, $Th_{maxHwUtil}(CPU)$ =40%); *UnbalancedLoad Rule* - the PCM Active Resource CPU of $Proc_4$ has an utilisation of 0.7% (i.e. lower than the threshold value, $Th_{minHwUtil}(CPU)$ =10%).
...	...

the second column instantiates the *problem* by reasoning on the PCM model elements. In particular, the application of the detection rules (e.g. QueueLength, see Section 4.1) is shown and the numerical value of some thresholds is reported (e.g. the upper and lower bounds for CPU devices are respectively set to 40% and 10%).

Note that several instances of the same antipattern type can be detected. For example, we found five instances of the Concurrent Processing Systems (not shown in Table 12 for sake of space) in the BRS system. Such antipatterns instances are not independent since two of them (i.e. PA_1 and PA_5) contain the CPU of $Proc_2$ as the over utilised one, whereas the remaining three instances (i.e. PA_8, PA_9 and PA_{10}) contain the HardDisk of $Proc_3$ as the over utilised one.

6.2.2. Ranking Antipatterns

Table 13 contains the performance requirement under study: it requires that the response time of the system is not larger than 27.5 seconds, whereas the performance analysis reveals a response time of 46.34 seconds. The *violated requirement* we consider covers the whole software model, in fact the involved entities column of Table 13 includes all the BRS model elements.

The solvable performance antipatterns, i.e. the ones that we consider for the ranking and the solution (see shaded labels of Figure 6), are collected in the *complete antipatterns list*, as shown in Table 14.

The *ranked antipatterns list* is reported in Table 15: it represents the result of our antipatterns ranking process, where numerical values are calculated according to the equations reported in Section 4.2. Note that the

Table 13: BRS - Violated Requirements.

ID	Requirement	Required Value	Observed Value	Involved Entities
R_1	RT(*system*)	27.5 sec	46.34 sec	*Webserver*, $Proc_1$, *Scheduler*, *UserManagement*, *OnLineReporting*, *GraphicalReporting*, $Proc_2$, *Database*, $Proc_3$, *Cache*, *CoreReportingEngine*, $Proc_4$

Table 14: BRS - Complete Antipatterns List.

ID	Detected Antipattern	Involved Entities	Contributing Indices
PA_1	CPS	$Proc_1$	U($Proc_1$) = 0.05, U($Proc_2$) = 0.49
PA_4	OLB	Database	U(*DBConnection*) = 0.68, U($Proc_3$) = 0.45
PA_5	CPS	$Proc_4$	U($Proc_2$) = 0.49, U($Proc_4$) = 0.07
PA_7	EP	*IOnlineReporting*. *onlineReport*, *GraphicalReporting*. *graphicalReport*	T(*onlineReport*) = 1.78, T(*graphicalReport*) = 0.42
PA_8	CPS	$Proc_1$	U($Proc_1$) = 0.05, U($Proc_3$) = 0.45
PA_9	CPS	$Proc_2$	U($Proc_2$) = 0.49, U($Proc_3$) = 0.45
PA_{10}	CPS	$Proc_4$	U($Proc_3$) = 0.45, U($Proc_4$) = 0.07

most guilty antipattern for the $RT(system)$ requirement is PA_4 whose guilt degree is 1.105, whereas the least guilty one is PA_9 with a guilt degree of 0.192. The solution of the detected antipatterns gives rise to software model candidates that we progressively numerate as BRS_{1-j} because they belong to the first iteration. Hence, the solution of PA_1 gives rise to the software model candidate BRS_{1-1}, the solution of PA_4 gives rise to the software model candidate BRS_{1-2}, and so on up to the solution of PA_{10} that gives rise to the candidate BRS_{1-7}.

In order to validate the efficiency of our ranking process we anticipate the first iteration of the antipattern-based approach by solving all the detected antipatterns.

The results of the software model *candidates* (i.e., $BRS_{PA_{1-1}}$, ..., $BRS_{PA_{1-7}}$, the candidates obtained by solving all the detected antipatterns) are collected in Table 16. It can be noticed that the high guilt degree of PA_4 has provided a relevant information because its removal improves the response time the most (from 46.34 seconds to 33.8 seconds). Similarly, the low guilt degree of PA_9 has provided relevant information too, in fact its removal makes worse the observed value for the requirement (from 46.34 seconds to 73.44 seconds), hence its solution affects much less the requirement.

In Figure 7 we summarise our experimentation to as-

Table 15: BRS - Ranked Antipatterns List.

Requirement	Detected Antipatterns						
	PA_1	PA_4	PA_5	PA_7	PA_8	PA_9	PA_{10}
R_1	0.445	1.105	0.539	0.812	0.406	0.192	0.500

Table 16: The requirement RT(*system*) across different software model candidates.

Requirement	Observed Values							
	BRS	BRS_{1-1}	BRS_{1-2}	BRS_{1-3}	BRS_{1-4}	BRS_{1-5}	BRS_{1-6}	BRS_{1-7}
R_1	46.34 sec	44.29 sec	33.8 sec	43.77 sec	44.61 sec	47.91 sec	73.44 sec	47.35 sec

sess the efficiency of the ranking process for the requirement under study. The target performance index (i.e. the response time of the system) is plotted on the y-axis, whereas on the x-axis the degree of guilt for antipatterns is represented. Single points represent the response times observed after the separate solution of each performance antipattern, and they are labeled with the ID of the antipattern that has been solved for that specific point. Of course, the points are situated, along the x-axis, on the corresponding guilt degree of the specific antipattern.

What is expected to observe in such representation is that the response time decreases while increasing the guilt degree of antipatterns, that is while moving from left to right on the diagram. In fact, in our experimentation we can observe a correlation of response time and guilt degree: Figure 7 shows that points with high guilt degree indeed have a lower response time. This confirms that solving a more guilty antipattern helps much more than solving a less guilty one, thus validating our guilt metric.

6.2.3. Solving Antipatterns

Table 17 reports some examples of the solved antipatterns for the BRS software model in the first iteration. Two columns are defined: the first one indicates the *antipattern* type according to the Smith and Williams classification [12]; the second one instantiates the *solution* by reasoning on the PCM model elements. In particular, the application of the refactoring actions (e.g. UnblockExecution, see Section 4.3) is shown.

Note that several instances of the same antipattern type can be solved. For example, the solution of the PA_1 antipattern suggests to re-deploy the *GraphicalReporting* component from $Proc_2$ to $Proc_1$, whereas the solution of the PA_5 antipattern suggests to re-deploy the same component from $Proc_2$ to $Proc_4$. It is for this reason that we consider refactoring actions separately, in order to avoid infeasible architectural alternatives.

Table 17: BRS- examples of solved antipatterns.

Antipattern	Solution
PA_7 - Extensive Processing	*UnblockExecution Action* - the scheduling algorithm of $Proc_2$ is changed from FCFS to PROCESSOR_SHARING.
PA_1 - Concurrent Processing Systems	*MostCritical Action* - the *GraphicalReporting* component is redeployed from $Proc_2$ to $Proc_1$.
PA_5 - Concurrent Processing Systems	*MostCritical Action* - the *GraphicalReporting* component is redeployed from $Proc_2$ to $Proc_4$.
PA_4 - One-Lane Bridge	*IncreaseCapacity Action* - the capacity of the passive resource of $Proc_3$ is increased by 5.
...	...

Each refactoring action from Table 17 results in a new software model candidate whose performance analysis reveals if the action is actually beneficial for the system under study. We recall that the solution of an antipattern cannot guarantee performance improvements in advance because the whole process is based on heuristics.

6.3. Further iterations of the antipattern-based process

To validate the efficiency of our approach the experimentation has been conducted as follows. A first analysis has been executed by applying the antipattern-based process without the ranking step: all the detected antipatterns are solved. A second analysis has been executed by introducing the ranking process with the *percentage* criterion (see Section 5), i.e. 40% of the detected antipatterns are discarded, i.e. the ones with the lowest degrees.

Figure 8 reports our experimentation by applying the antipattern-based process without the ranking step, and

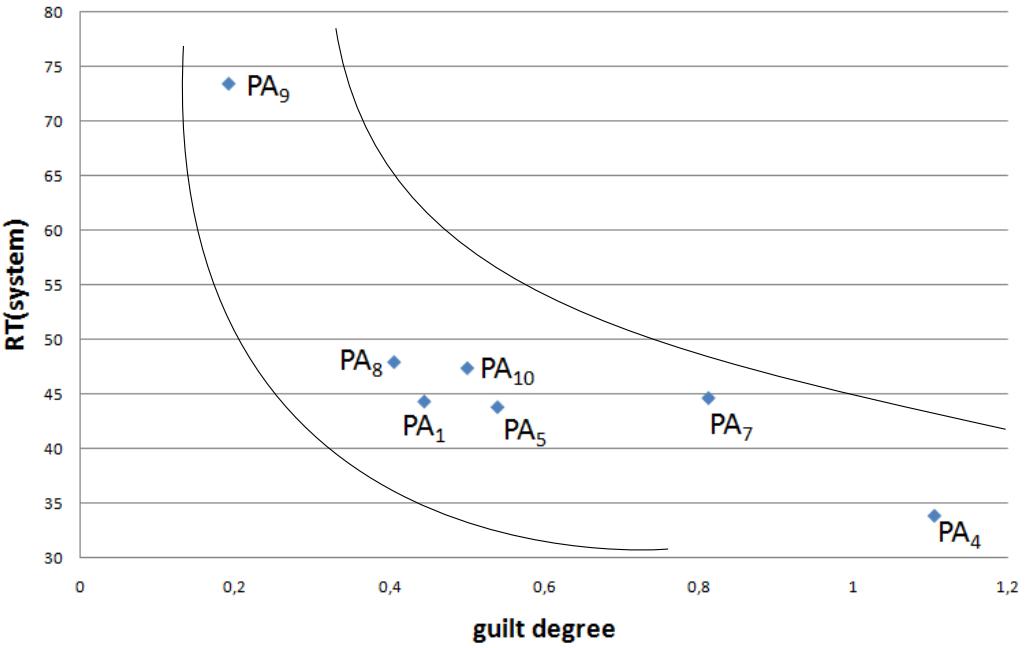

Figure 7: RT(*system*) vs the guilt degree of antipatterns.

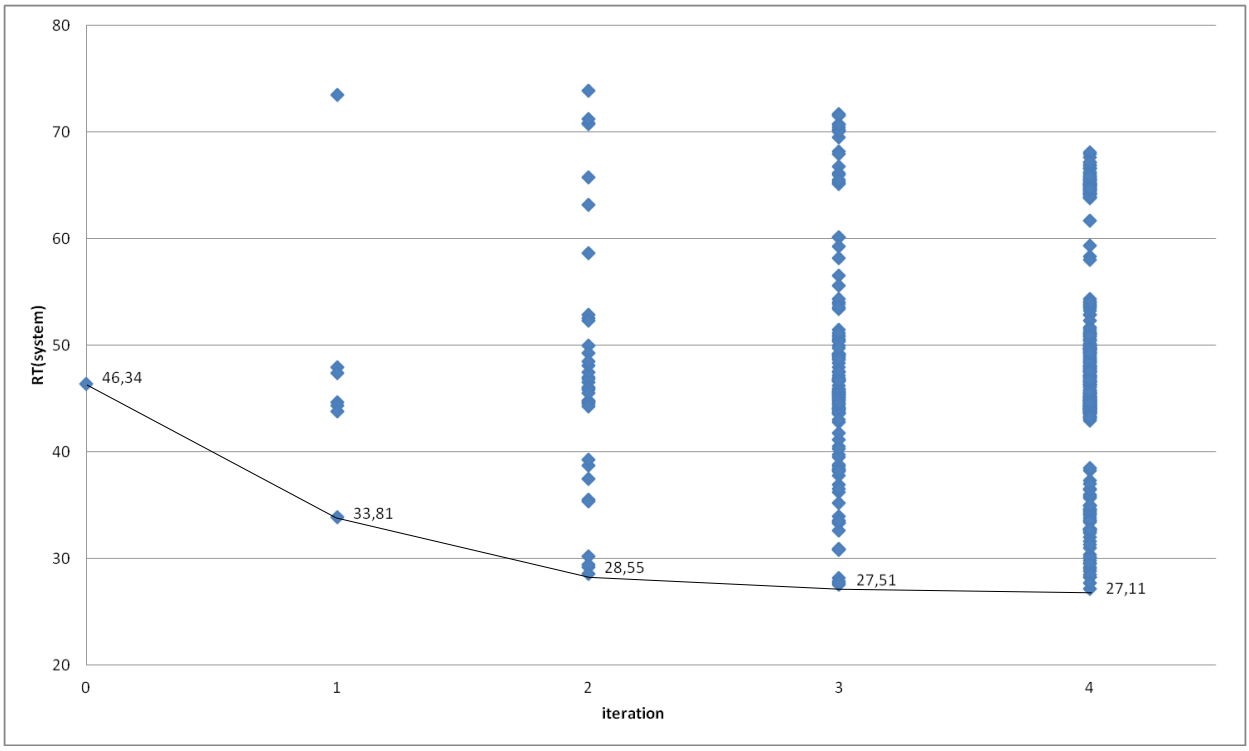

Figure 8: Response time of the *system* across the iterations of the antipattern-based process.

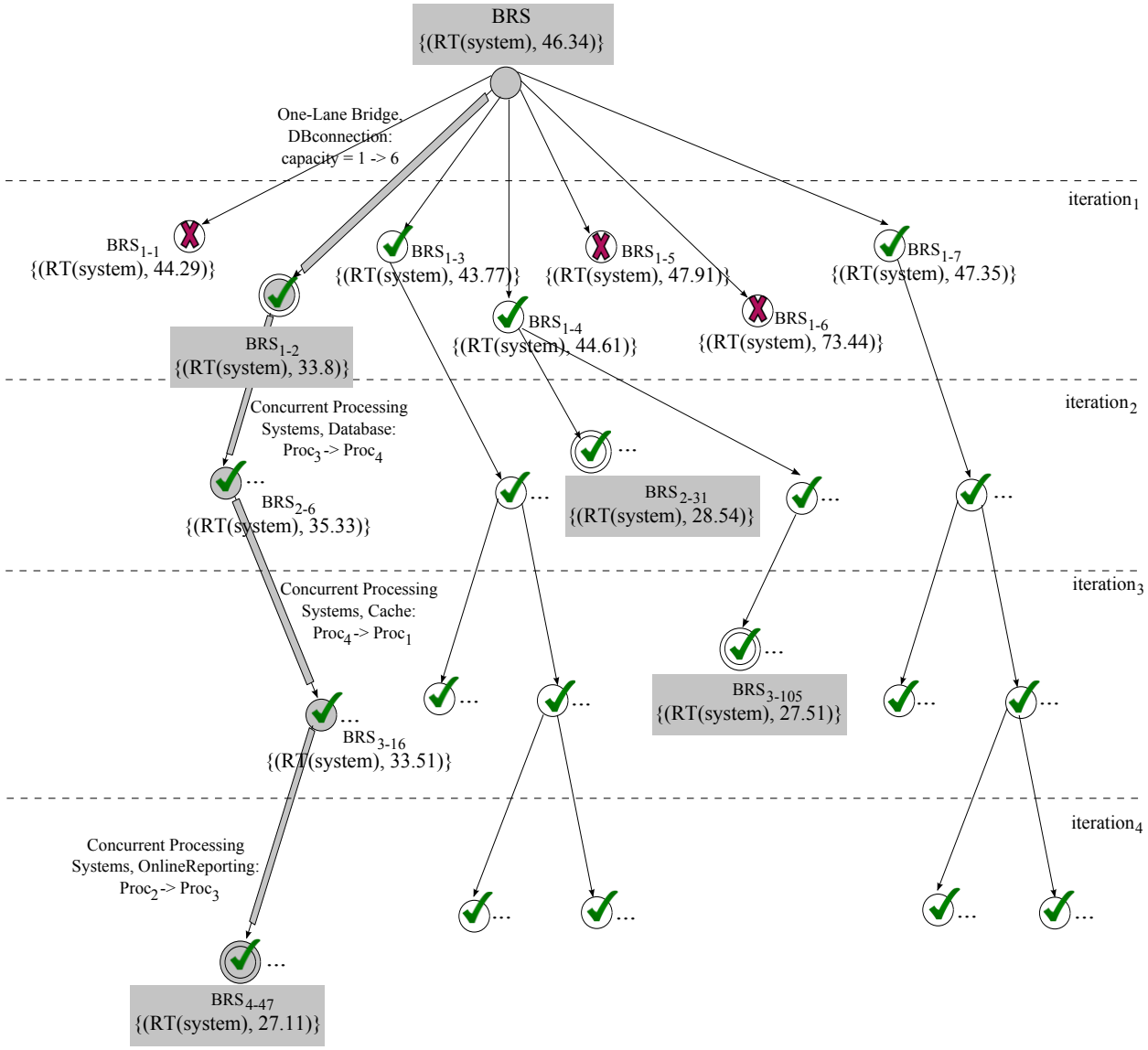

Figure 9: Process summary for the BRS system.

Table 18: BRS - reduction of the number of software model candidates by means of the ranking process.

	Number of Software Model Candidates				
Approach	$iteration_1$	$iteration_2$	$iteration_3$	$iteration_4$	TOT
Solution	7	36	123	289	455
Guilt-based Solution	4	25	93	217	339

across multiple iterations of the process: the target performance index is the response time of the *system* and it is plotted on the y-axis, while the iterations of the antipattern-based process are listed on the x-axis. Single points represent the response times observed after the separate solution of each performance antipattern.

Figure 8 summarizes the whole experimentation across the different *iteration*s: the response time of the system spans from 46.34 seconds (i.e. the initial value) to 27.11 sec (i.e. the value that fits with the requirement). Note that at each iteration a performance improvement is achieved up to the fourth iteration, and the final improvement is roughly of 45%.

Figure 9 reports our experimentation by applying the antipattern-based process with the ranking step, and the graph-like notation (see Section 5) summarizes the process: each node reports the performance index of our interest, i.e. RT(system), and its observed value (e.g. 46.34 seconds in the root of the graph represents the observed value for the initial system); each arc represents a refactoring *action* (e.g. capacity of the passive resource DBconnection is increased from 1 to 6) applied to solve a detected *antipattern* (e.g. One-Lane Bridge). In our experimentation we applied the *fulfilment criterion* (see Section 5) to terminate the process, since the requirement is satisfied at the fourth iteration and a software model candidate (i.e. BRS_{4-47}, see Figure 9) able to cope with user needs is found.

Note that the ranking process has been executed only on the initial candidate, i.e. *BRS* software model. Figure 9 reports all the candidates of the first iteration, i.e. BRS_{1-1}, ..., BRS_{1-7}, and their observed values for the requirement under study, as anticipated in Table 16.

According to the defined selection criterion, all the design alternatives coming from the 40% of detected antipatterns with the lowest guilt degree are discarded. As shown in Table 15, the lowest guilt degrees come from the antipatterns PA_1, PA_8 and PA_9, hence the BRS_{1-1}, BRS_{1-5} and BRS_{1-6} software model candidates are discarded, i.e. the nodes with the $\times$ symbol in Figure 9.

The experimental results we obtained are finally collected in Table 18 where the two approaches are compared. If we apply the *solution* step without the support of the ranking process 455 software model candidates are generated: 7 in the first iteration, 36 in the second iteration, 123 in the third iteration, and 289 in the fourth iteration. When introducing the ranking process the antipattern-based approach roughly discards the 34% of design alternatives: if we apply the *guilt-based solution* step in the first iteration, only 339 software model candidates are analysed.

In our experimentation the ranking process supported the antipatterns' solution step by converging towards the desired performance improvement, and without compromising the path towards the final best candidate. Figure 9 additionally shows that at the second iteration the local optimum is achieved at the 31st candidate (i.e. BRS_{2-31}) whose performance analysis reveals a response time for the system equal to 28.54 seconds, whereas at the third iteration the local optimum is achieved at the 105th candidate (i.e. BRS_{3-105}) whose performance analysis reveals a response time for the system equal to 27.51 seconds. Although both these candidates do not belong to the global optimum path, they are not discarded by the applied ranking methodology. The fourth iteration produces a candidate whose response time is equal to 27.11 seconds, hence it satisfies the requirement and the process terminates.

The number of necessary iterations obviously depends on the stopping criterion that, in this case, was the one driven by a requirement (i.e. *fullfillment* criterion). We have explicitly decided to not considering the relative improvement of performance between two iterations as a stopping criterion because, basing on our experience, the solution of one antipattern can surprisingly improve the system performance even on a system where recent actions have not brought substantial improvements. In fact, by observing the response time values of candidates in Figure 9, the statistical difference between one iteration and the next one does not represent a key factor for the global optimum search.

It is relevant to notice that in our experimentation the global optimum path includes a candidate node that has worse performance than its parent, in fact at the second iteration the 6th candidate (i.e. BRS_{2-6}) reveals a response time for the system equal to 35.33 seconds, i.e. larger than the one from which it is generated (i.e. BRS_{1-2}), and whose response time for the system is equal to 33.8 seconds.

Note that the selection criterion strongly influences the advantages of applying the ranking process. Other considerations can be done if we modify the percentage value (currently set to 40%) for discarding all the detected antipatterns. While discarding one-by-one the remaining antipatterns, i.e. PA_4, PA_7, PA_5, PA_{10}, (from the less promising, i.e. PA_{10}, up to the most promising, i.e. PA_4) we can observe the following numbers of software model candidates: 255, 162, 114. Hence, the ranking methodology may benefit the whole antipattern-based process, since we experimented a relevant reduction in the number of model candidates.

As shown in Figure 9, we can conclude that the software model candidate that best fits with user needs is obtained by applying the following refactoring actions

(see the shaded path of Figure 9): (i) the capacity of the passive resource DBconnection is increased from 1 to 6; (ii) the Database component is redeployed from $Proc_3$ to $Proc_4$; (iii) the Cache component is redeployed from $Proc_4$ to $Proc_1$; (iv) the OnlineReporting component is redeployed from $Proc_2$ to $Proc_3$.

7. The approach at work on other three case studies

In this Section we report our experimentation on three different case studies to demonstrate the effectiveness of the antipattern-based process, and it is organised as follows. Section 7.1 describes the PCM models of the three systems under analysis. Section 7.2 reports the experimental results and demonstrates that the introduction of the ranking methodology in the solution step leads the system to converge towards the desired performance improvement while discarding a substantial number of design alternatives.

7.1. PCM models of case studies

In this Section we describe the models of three systems used as testbed for our antipattern-based process. These systems are aimed at consolidating our confidence on the usability of the approach. They have been modeled by three groups of graduate students of the Advanced Software Engineering course at the University of L'Aquila, as part of their homeworks. The three domains represented the cores of the course in the last three years (one per year). Thus, for sake of our experimentation we have selected the best homework for each year (hence for each domain). Furthermore, one of these homeworks has been awarded by a Business-Plan Competition [38]. Even if they are not really implemented systems, they have been exposed in research projects as realistic and characteristic for their domains [39].

7.1.1. E-Commerce System

The system under study is the so-called E-Commerce System (ECS) that is a web-based system managing business data: customers browse books and movies catalogues and make selections of items that need to be purchased.

Figure 10 shows an overview of the PCM software model for the ECS system. The system supports four use cases: customers can register, login, and request to browse catalogues and to purchase some products. The *WebServer* component is connected to: (i) the *CustomerController* component that manages customers' requests, such as login and registration, together with the *Database* component; (ii) the *BooksDispatcher* and *MoviesDispatcher* components that manage the products offered by ECS. In particular, the customers are managed by the *CustomerController* component that communicates with the *Database* to: (i) store the registration of new users, i.e. their profile, preferences and pictures; (ii) verify the login credentials of users already registered. The *BooksDispatcher* component communicates with the *BooksController* component that retrieves information (such as the availability and the price of the products) from the *Database* component. Similarly, the *MoviesDispatcher* communicates with the *MoviesController* component that retrieves information from the *Database* component. Both *BooksController* and *MoviesController* components communicates with the *BooksCatalog* and the *MoviesCatalog* respectively. The allocation of software components on resource containers is shown in Figure 10, e.g. the $DispatcherNode$ hosts the *CustomerController*, *BooksDispatcher*, and *MoviesDispatcher* basic components.

The PCM software model additionally contains the usage model specifying how users use the system: users make the registration, then they login, 5 times the *browseCatalog* service is invoked, and finally the *makePurchase* service is executed.

The performance analysis of the ECS software model reveals that the response time of the system is 76.96 seconds (under the expected workload of 10 requests per second with thinking time of 5 seconds), so it does not meet the stated requirement, i.e. 25 seconds. Since the requirement is not satisfied, we apply our approach to detect, rank and solve performance antipatterns. Experimental results are discussed in Section 7.2.

7.1.2. E-Health System

The system under study is the so-called E-Health System (EHS) that is a web-based system supporting the doctors' and patients' everyday activities. Doctors are allowed to retrieve the information of their patients and, on the basis of such data, they can send an alarm in case of warning conditions. Patients are allowed to retrieve information about the doctor expertise and update some vital parameters, e.g. heart rate, that are required to monitor their health status.

Figure 11 shows an overview of the PCM software model for the EHS system by reporting the software components and their dependencies. *PatientDispatcher* and *DoctorDispatcher* components are connected to the *Scheduler* component that firstly checks the user credentials by communicating with the *UserManagement* component and then forwards users' requests to the *DataManagement* component. This latter component communicates with *DatabaseData* component and/or

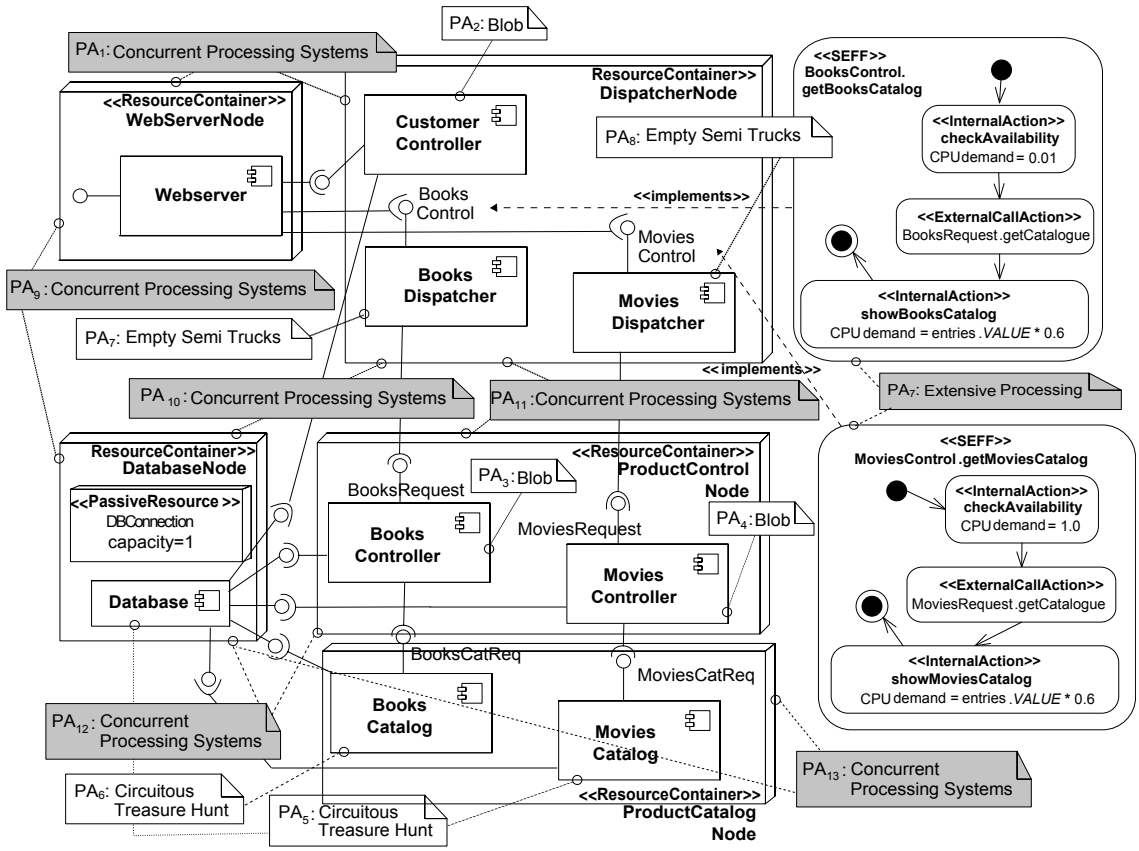

Figure 10: PCM Software Model for the ECS system.

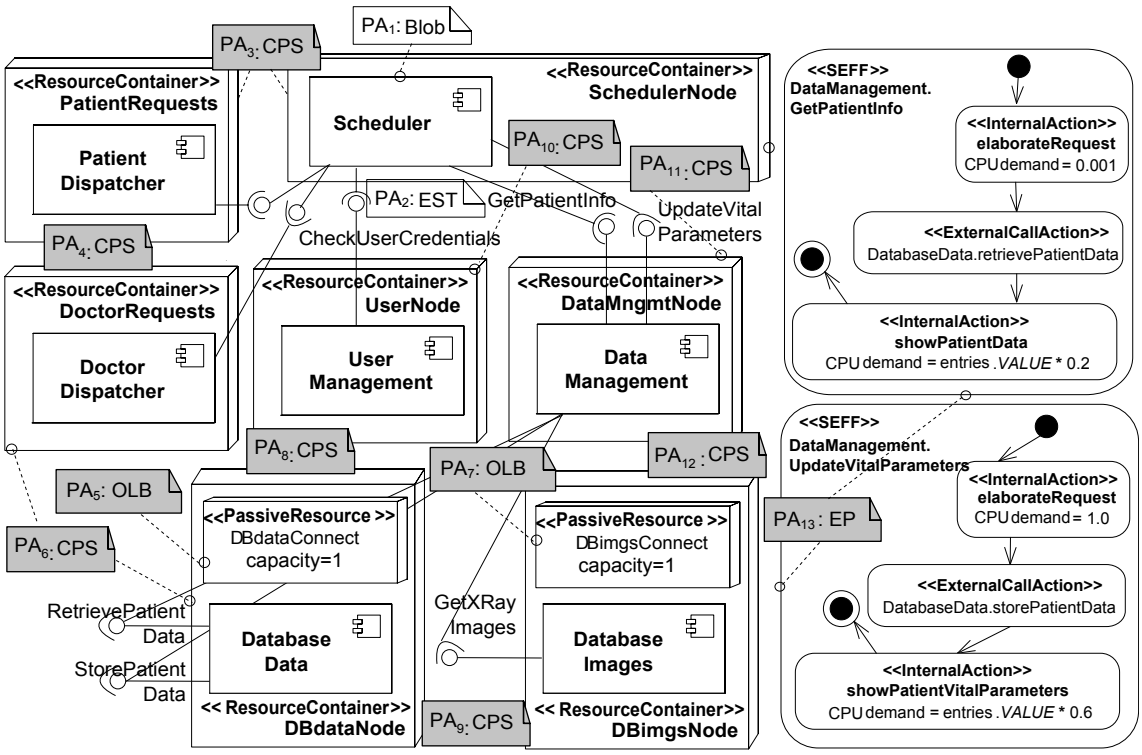

Figure 11: PCM Software Model for the EHS system.

retrieves images from the *DatabaseImages* component. In particular, the *DataManagement* component provides two services: (i) the *GetPatientInfo* service is aimed at retrieving patient data and send it to the doctor's application; (ii) the *UpdateVitalParameters* service is aimed at storing patient data in the database and send back the acknowledgment of such operation to the patient's application. The allocation of software components on resource containers is shown in Figure 11, e.g. the $SchedulerNode$ hosts the *Scheduler* basic component, the $UserNode$ hosts the *UserManagement* basic component, etc.

The PCM software model additionally contains the usage model specifying how users use the system: doctors invoke the *Login* and the *GetPatientInfo* services, and in case of warning conditions (regulated by a probability of 20%) they send an alarm, whereas patients invoke the *UpdateVitalParameters* service.

The performance analysis of the EHS software model reveals that the response time of the system is 4.26 seconds (under the expected workload of 30 requests per second with thinking time of 10 seconds), so it does not meet the stated requirement, i.e. 3 seconds. Since the requirement is not satisfied, we apply our approach to detect, rank and solve performance antipatterns. Experimental results are discussed in Section 7.2.

7.1.3. Bus On Air System

The system under study is the so-called Bus on Air System (BOA) that is a system aimed at developing a set of services for public transportation targeting both the end-users, such as passengers, and the suppliers, such as transportation agencies.

Figure 12 shows an overview of the PCM software model for the BOA system. Each request of passengers is represented by the *MobileApplication* and it is forwarded to *Balancer* component. Requests are managed by a *Server* component that retrieves data from two *Database* components and send data back to the mobile applications. Two services have been specified in the system: (i) *GetConnections*, i.e. the potential passenger arrives at the bus stop and has access to information on the best path to reach a destination, such as lines that cover a path, how long to wait, etc.; (ii) *GetFacilities*, i.e. the potential passenger arrives at the bus stop and has access to information on the facilities nearby its current location, such as bars, fast foods, shops, ATMs, attractions, how far they are located, etc. Two different databases have been defined to access data information: (i) the *DatabaseConnections* component retrieves information about the best path to reach a destination place and the public transportation that can be used by passengers; (ii) the *DatabaseFacilities* component retrieves information about the facilities (e.g. fast foods) nearby the current location provided by passengers. The allocation of software components on resource containers is shown in Figure 12, e.g. the *BalancerNode* hosts the *Balancer* basic component, the $ServerNode$ hosts the *Server* basic component, etc.

The PCM software model additionally contains the usage model specifying how users use the system: users invoke the *GetConnections* service, and in case they need to wait for long (regulated by a probability of 70%) they invoke the *GetFacilities* service by which they discover the nearby facilities.

The performance analysis of the BOA software model reveals that the response time of the *GetConnections* service is 9.51 seconds (under the expected workload of 15 requests per second with thinking time of 3 seconds), so it does not meet the stated requirement, i.e. 2 seconds. Since the requirement is not satisfied, we apply our approach to detect, rank and solve performance antipatterns. Experimental results are discussed in Section 7.2.

7.2. Quantifying the benefit of ranking in the antipattern-based approach

In this Section we discuss the experimental results of the three different case studies presented in Section 7.1.

The antipattern-based approach is applied by distinguish three different methodologies: (i) solution, i.e. no ranking methodology is considered in the step of solving antipatterns; (ii) guilt-based solution (without semantic factor), i.e. the ranking methodology does not consider the ranking refinement; (iii) guilt-based solution (with semantic factor), i.e. the guilt degree of antipatterns takes into account the specification of our ranking refinement.

In these case studies we adopt the ranking methodology with the *limit* selection criterion (see Section 5), hence we discard all the detected antipatterns whose guilt degree is lower than a numerical limit value. In particular, the (ii) guilt-based solution (without semantic factor) is evaluated by discarding all antipatterns whose guilt degree is lower than a 0.4 limit value, whereas the (iii) guilt-based solution (with semantic factor) is evaluated by discarding all antipatterns whose guilt degree is lower than a 0.5 limit value.

This difference in the value of the *limit* selection criterion is due to the different estimation for the guilt degree of antipatterns. In fact, as said in Section 4.2, the guilt degree is calculated giving a score to each involved en-

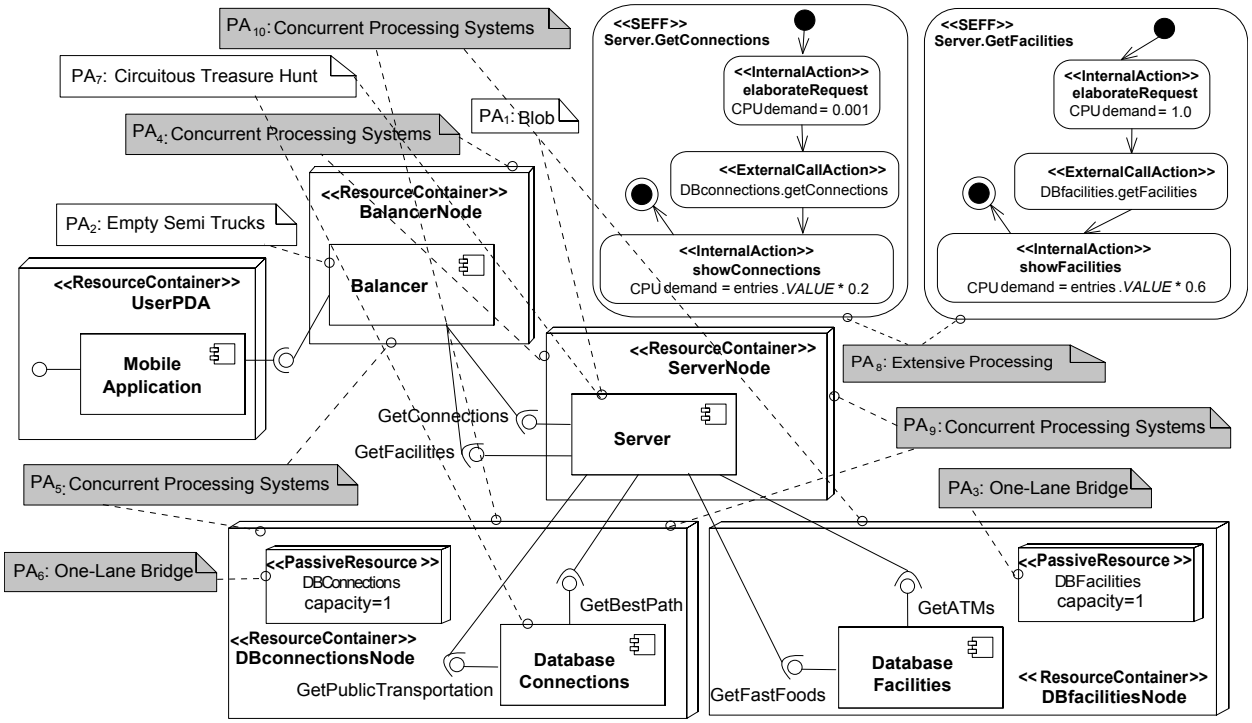

Figure 12: PCM Software Model for the BOA system.

tity as well as the semantic factor that varies in the interval $[0, \ldots, 1]$. Hence the guilt degree of each antipattern varies in the interval $[0, \ldots, n]$ (where n is the number of involved entities in the violated requirement) without the semantic factor and in the interval $[0, \ldots, n+1]$ when the calculation takes into account the semantic factor. Overall, the semantic factor is aimed at capturing the potential effectiveness of an antipattern solution by considering additional information. Hence by moving the *limit* from 0.4 to 0.5 we aim at discarding all the antipatterns whose guilt degree does not embed at least 25% of additional information brought by the semantic factor.

Table 19 demonstrates the benefit of the ranking process by quantifying the reduction of the number of software model candidates for each considered case study. Experimental results are discussed in the following.

E-Commerce System. In our experimentation we found that after two iterations of the antipattern-based process the response time decreases from 76.96 seconds to 20.73 seconds, thus achieving a performance improvement of 73%. The software model candidate that best fits with user needs is obtained by applying the following refactoring actions: (i) the scheduling algorithm of *DispatcherNode* is modified from *FCFS* to *PROCESSOR_SHARING*; (ii) the *MoviesCatalog* component is redeployed from *ProductCatalogNode* to *DispatcherNode*.

Table 20 reports additional information about ten randomly chosen software model candidates. For each candidate we report: the lower and upper bounds of the 95% confidence interval of the system response time, the size of the interval, as well as the response time mean value. The shaded row represents a candidate that satisfies the user needs.

We remark that the interval size is always well under the 1% of its lower bound. This implies a very small variance of these values around their mean value, and a very high confidence in these simulation results when they are used within the antipattern-based process. For sake of completeness, we have also calculated the average interval size overall the analysed candidates for this case study, and we have obtained a value of 0.0302 sec. This consolidates our confidence in the usage of mean values within the whole process.

Table 19 shows that if we apply the *solution* step without the support of the ranking process 225 software model candidates are generated: 13 in the first iteration, and 212 in the second iteration. When introducing the ranking process without the semantic factor the antipattern-based approach roughly discards 45% of de-

Table 19: Other case studies - reduction of the number of software model candidates by means of the ranking process.

	Number of Software Model Candidates											
	E-Commerce System				E-Health System				Bus On Air System			
Approach	$iteration_1$	$iteration_2$	TOT	*saving* (%)	$iteration_1$	$iteration_2$	TOT	*saving* (%)	$iteration_1$	$iteration_2$	TOT	*saving* (%)
Solution	13	212	225	-	19	244	263	-	11	126	137	-
Guilt-based Solution (without semantic factor)	12	111	123	45%	11	82	93	64%	8	56	64	53%
Guilt-based Solution (with semantic factor)	6	48	54	76%	9	54	63	76%	5	27	32	77%

Table 20: E-Commerce System: 95% confidence interval of response time (in seconds).

Lower Bound	Mean Value	Upper Bound	Interval Size
76.9378	76.9632	76.9845	0.0467
31.9877	32.0027	32.0069	0.0192
25.7303	25.7489	25.7506	0.0203
124.3860	124.4251	124.4662	0.0802
25.7198	25.7357	25.7389	0.0191
54.7712	54.7849	54.7994	0.0282
20.7303	20.7329	20.7406	0.0103
29.3585	29.4107	29.4433	0.0848
25.6793	25.6951	25.6983	0.0190
32.0694	32.0828	32.0898	0.0204

sign alternatives, in that only 123 software model candidates are analysed. Furthermore, the usage of the semantic factor leads to roughly discard 76% of original design alternatives, in that only 54 software model candidates are analysed.

E-Health System. In our experimentation we found that after two iterations of the antipattern-based process the response time decreases from 4.26 seconds to 2.69 seconds, thus achieving a performance improvement of 37%. The software model candidate that best fits with user needs is obtained by applying the following refactoring actions: (i) the capacity of the passive resource *DBdataConnect* is increased by 5 (i.e. from 1 to 6); (ii) the *DatabaseData* component is redeployed from *DBdataNode* to *SchedulerNode*.

Similarly to Table 20, Table 21 reports additional information about ten randomly chosen software model candidates for the E-Health System. In this case the interval size is under the 3% of its lower bound, and this implies a small variance around the mean value. Furthermore, the average interval size overall the analysed candidates is equal to 0.0504 sec. This consolidates our confidence in the usage of mean values within the whole process for this case study as well.

Table 21: E-Health System: 95% confidence interval of response time (in seconds).

Lower Bound	Mean Value	Upper Bound	Interval Size
4.2445	4.2612	4.2953	0.0508
3.9565	4.0345	4.0449	0.0884
3.2203	3.2944	3.3158	0.0955
3.8726	3.9456	3.9735	0.1009
3.9189	3.9846	4.0163	0.0974
2.6801	2.6881	2.6911	0.0110
4.1539	4.2309	4.2711	0.1172
4.0416	4.1184	4.1479	0.1063
3.8372	3.9195	3.9551	0.1179
4.0668	4.1448	4.1697	0.1029

Table 19 shows that if we apply the *solution* step without the support of the ranking process 263 software model candidates are generated: 19 in the first iteration, and 244 in the second iteration. When introducing the ranking process without the semantic factor the antipattern-based approach roughly discards 64% of design alternatives, in that only 93 software model candidates are analysed. Furthermore, the usage of the semantic factor leads to roughly discard 76% of original design alternatives, in that only 63 software model candidates are analysed.

Bus On Air System. In our experimentation we found that after two iterations the response time decreases from 9.51 seconds to 1.48 seconds, thus achieving a performance improvement of 84%. The software model candidate that best fits with user needs is obtained by applying the following refactoring actions: (i) the scheduling algorithm of *ServerNode* is modified from *FCFS* to *PROCESSOR_SHARING*; (ii) the capacity of the passive resource *DBConnections* is increased by 5 (i.e. from 1 to 6).

Similarly to Tables 20 and 21, Table 22 reports additional information about ten randomly chosen software model candidates for the Bus On Air System. In this case the interval size is under the 2.5% of its lower bound, and this implies a small variance of these values around their mean value, thus to get a high confidence in these simulation results when they are used within the antipattern-based process. We have also calculated the average interval size overall the analysed candidates for this case study, and it is equal to 0.0498 sec.

Table 22: Bus On Air System: 95% confidence interval of response time (in seconds).

Lower Bound	Mean Value	Upper Bound	Interval Size
9.4929	9.5134	9.5268	0.0339
2.9917	3.0049	3.0205	0.0288
2.9311	2.9404	2.9507	0.0196
4.0009	4.0224	4.0514	0.0505
1.4647	1.4843	1.4992	0.0345
4.9580	5.0016	5.0718	0.1138
2.7299	2.7471	2.7604	0.0305
2.9915	3.0055	3.0190	0.0275
4.9528	4.9630	4.9740	0.0212
2.7601	2.7797	2.7963	0.0362

Table 19 shows that if we apply the *solution* step without the support of the ranking process 137 software model candidates are generated: 11 in the first iteration, and 126 in the second iteration. When introducing the ranking process without the semantic factor the antipattern-based approach roughly discards 53% of design alternatives, in that only 64 software model candidates are analysed. Furthermore, the usage of the semantic factor leads to roughly discard 77% of original design alternatives, in that only 32 software model candidates are analysed.

Figure 13 reports the guilt degree values of the detected antipatterns for the BOA system, where the x-axis represents the analysed antipatterns and the y-axis the guilt degree associated to each antipattern and calculated without the semantic factor. The horizontal line in the figure denotes the *limit* selection criterion (i.e. 0.4) that splits the figure in two parts: all points below the line represent the antipatterns whose guilt degree does not fulfil the selection criterion and are discarded by our approach, whereas all points above the line represent the ones that fulfil the criterion. These latter points represent 64 detected antipatterns that are further evaluated by our approach (see Table 19).

Figure 14 reports similar results where the semantic factor is taken into account. The horizontal line in the figure denotes the *limit* selection criterion of this case (i.e. 0.5). Again, the points above the line represent 32 detected antipatterns that are further evaluated by our approach (see Table 19).

Table 23 reports the number of software model candidates while varying the limit values for the selection criterion associated to the ranking process. Latter values are reported in the first table row as *x* / *y*, where *x* and *y* represent the values used by the guilt-based solution, without and with semantic factor respectively.

We can notice that obviously, while decreasing the limit values for the selection criterion, the number of software model candidates increase. In the Bus On Air System we found that if we set the limit value to 0.6/0.7 then only one candidate is selected by using the guilt-based solution (without semantic factor) and no candidates survive to the guilt-based solution (with semantic factor). On the other end, if we set the limit value to 0.1/0.2 then 103 candidates are selected by using the guilt-based solution (without semantic factor) and 56 candidates by the guilt-based solution (with semantic factor).

The shaded values in Table 23 corresponds to the ones reported in Table 19 where the limit selection criteria were chosen as equal to 0.4 and 0.5, respectively. This choice, that has been applied to all case studies, comes from the need of considering sets of candidates, on one side, sufficiently large to provide enough alternatives and, on the other end, not too large to limit the whole process complexity.

Table 23: Number of selected software model candidates vs. the limit values of the ranking process for the Bus On Air System.

Limit values for the selection criterion	0.6/ 0.7	0.5/ 0.6	0.4/ 0.5	0.3/ 0.4	0.2/ 0.3	0.1/ 0.2
Guilt-based Solution (without semantic factor)	1	17	64	89	96	103
Guilt-based Solution (with semantic factor)	0	11	32	49	56	56

Summarizing, in our experimentation the ranking process supported the antipatterns' solution step by converging towards the desired performance improvement, and without compromising the path towards the final best candidate. In the considered case studies we found that the introduction of the ranking methodology greatly benefits to the process, between 45% and 64%. Furthermore, the introduction of the semantic factor additionally benefits to the ranking methodology by reducing the number of software model candidates up to 77%.

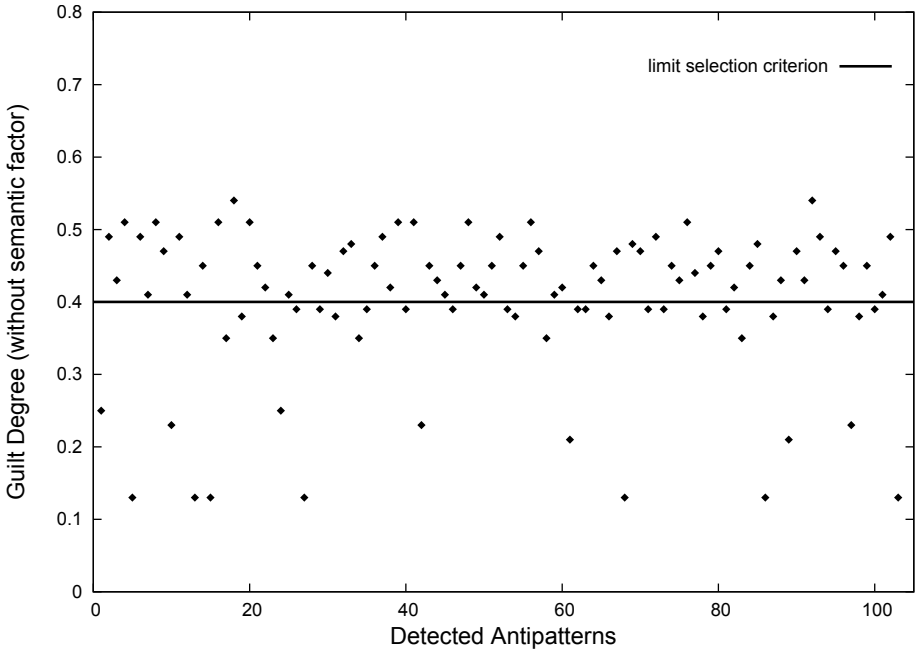

Figure 13: Bus On Air System: Guilt Degree of Detected Antipatterns (without semantic factor).

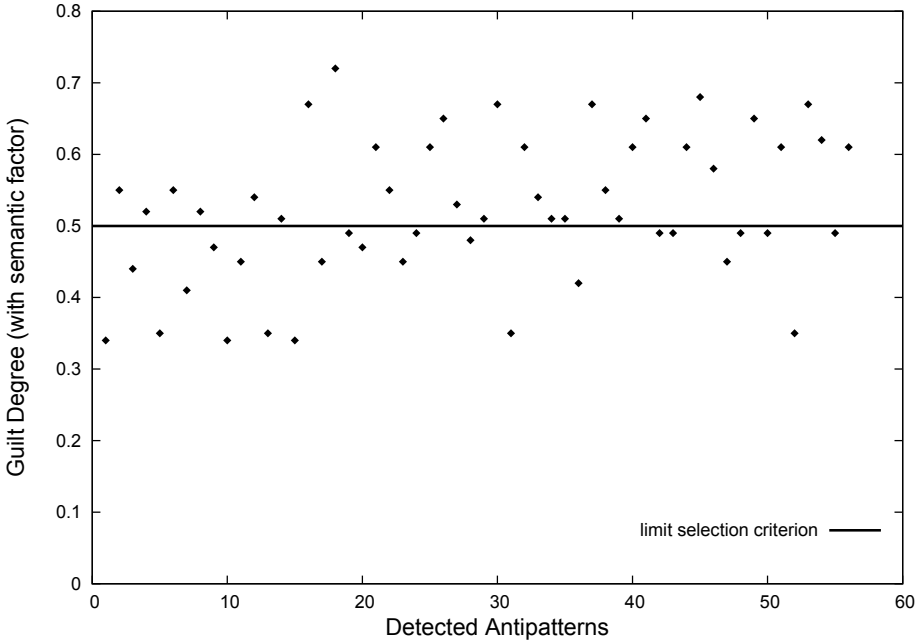

Figure 14: Bus On Air System: Guilt Degree of Detected Antipatterns (with semantic factor).

8. Discussion

The approach presented in this paper highlights the complexity of the identification and removal of performance problems. In this section we focus on assumptions, limitations, and open issues that this work raises.

Approach validation. We have applied our approach to four case studies belonging to different application domains. The results that we have reported in the previous sections provide good evidence (by a quantitative viewpoint) of the process improvement that our ranking methodology, as well as the semantic factors, can lead. We retain that such an assessment performed on semantically meaningful case studies supports more appropriately the adequacy (and the transferability) of our approach with respect to a massive application of the approach to hundreds of automatically generated models. In fact, latter ones could not have any similarity to real domain examples, so weakening the experimentation.

Complexity of the detection algorithm and ranking benefits. The search algorithm is not geared for optimal performance, and no complexity analysis has been conducted. In average, the algorithm based on ranking has returned the list of detected antipatterns in few minutes. Of course, the net time saving of our approach depends on the complexity of the application model. In fact, the performance model solution could represent a heavy task to accomplish in case of quite complex models. This aspect represents an open issue, and we plan to investigate more on the algorithm complexity in order to evaluate its scalability.

Correctness of rules and actions. The rules and the actions we propose for detecting and solving antipatterns reflect our interpretation of the natural language definitions [12]. Other approaches might interpret and formalize the antipatterns differently. This unavoidable gap is an open issue in this domain, and certainly requires a wider investigation to consolidate the formal definition of antipatterns. Beside this, the threshold binding is another aspect that may affect the detection process. We have been recently working on this issue and our first results can be found in [36].

Human contribution. The process of refactoring software models cannot be fully automated, since some decisions are not machine-processable. Obstacles to the application of the refactoring actions can originate for different reasons, such as legacy constraints, budget limitations, functional requirements, etc. For example, while redeploying software components it may happen that no hardware machine is sufficiently idle to host other components. In this case the analysts may decide to add a new hardware machine (that may be costly) or try to apply alternative refactoring actions. Hence, we retain the human role very relevant in the whole process, because quite often the experience of analysts cannot be codifies in simple rules. Beside this, since this process is not aimed at supporting runtime evaluations, the elapsed time is not a key issue, thus the system stakeholders can benefit from offline evaluations based on the process results.

Ranking assumption. A current limitation of the ranking process is that control flow forks are not yet supported. To account for control flow forks, the computation of composite services needs to be adjusted: the equations need to be heuristically approximated because the computation of services with parallelism cannot be derived from the response time of the services alone. Additionally, more experience could lead to refine the antipattern priorities on the basis of the application domain, e.g. the "One-Lane Bridge" antipattern might be of particular interest in database-intensive systems.

Simulation errors. The proposed antipattern-based process may be affected by simulation errors. In fact the antipattern detection and ranking operational steps are based on performance simulation results obtained with a certain confidence (i.e., 95% in our case studies). Confidence intervals of performance indices coming from simulation results obviously induce confidence intervals on the guilt degrees, thus affecting the ranking and selection of antipatterns to solve. For the case studies that we have considered (see Section 7) these intervals were very narrow for performance indices, so we are confident that they cannot heavily affect the decision process. However, we intend to investigate further this aspect in future.

9. Conclusion

In this paper we have presented an approach, based on antipatterns, that aims at identifying performance flaws and removing them within PCM-based architectural models. We implemented the approach as an extension of the PCM Bench tool.

The process of solving antipatterns has been improved with respect to [6] by introducing a ranking methodology that identifies, among a set of detected antipatterns, the ones that mostly contribute to the violation of specific performance requirements.

Four case studies have been presented to demonstrate the validity of the approach: the introduction of our ranking process in the solution step led in all cases the system to converge towards the desired performance improvement with considerable saving of candidate alternatives.

Using our approach, performance analysts can detect and solve performance problems more quickly. Instead of manually analysing the result indices of performance analyses without coming up with possible design alternatives, they only have to assess the refactoring actions suggested by our antipattern-based approach.

The results we obtained by applying our antipattern-based approach are very good, however several open issues must be addressed in future within the PCM context as well as in a more general vision.

With regard to PCM, a key question is whether the automation of the yet unsupported antipatterns is useful in the PCM context. The introduction of more detailed modelling constructs to capture the antipattern properties, such as controller infrastructures, might lead to too high modelling efforts compared to the expected benefits. An accurate analysis must be conducted in order to evaluate the pros and cons of supporting the antipatterns that are currently not automated.

Other software modelling languages can be considered, too, if the concepts for representing antipatterns are available; for example, architectural description languages such as AADL [40] can be also suited to apply the approach. As future work, we plan to investigate the representation of antipatterns in different languages in order to gain experience for a more general framework, independent of any modelling notation.

We also plan to combine the antipattern-based process with multi-criteria evolutionary quality optimisation approaches, such as the one in [30]. Multi-criteria evolutionary quality optimisation tries to improve several quality attributes (such as performance and reliability) at once by iteratively evolving the software model, applying random mutation and crossover operators. Knowledge on performance antipatterns can be used to evolve candidates more effectively towards better performance.

Acknowledgments. This work was partially supported by the European Office of Aerospace Research and Development (EOARD), Grant Cooperative Agreement (Award no. FA8655-11-1-3055), and VISION ERC project (ERC-240555).

References

[1] C. M. Woodside, M. Franks, D. C. Petriu, The Future of Software Performance Engineering, in: Workshop on the Future of Software Engineering, 2007, pp. 171–187.

[2] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-based Performance Prediction in Software Development: A Survey, IEEE Transactions on Software Engineering 30 (5) (2004) 295–310.

[3] S. Bernardi, S. Donatelli, J. Merseguer, From UML sequence diagrams and statecharts to analysable petrinet models, in: Workshop on Software and Performance, 2002, pp. 35–45.

[4] C. M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, J. Merseguer, Performance by Unified Model Analysis (PUMA), in: Workshop on Software and Performance, 2005, pp. 1–12.

[5] C. U. Smith, L. G. Williams, Software Performance Antipatterns, in: Workshop on Software and Performance, 2000, pp. 127–136.

[6] C. Trubiani, A. Koziolek, Detection and solution of software performance antipatterns in palladio architectural models, in: WOSP/SIPEW International Conference on Performance Engineering, 2011, pp. 19–30.

[7] S. Becker, H. Koziolek, R. Reussner, The Palladio component model for model-driven performance prediction, Journal of Systems and Software 82 (2009) 3–22.

[8] S. Frølund, J. Koistinen, Quality-of-Service Specification in Distributed Object Systems, Tech. Rep. HPL-98-159, Hewlett Packard, Software Technology Laboratory (Sep. 1998).

[9] Q. Noorshams, A. Martens, R. Reussner, Using quality of service bounds for effective multi-objective software architecture optimization, in: International Workshop on the Quality of Service-Oriented Software Systems (QUASOSS), 2010, pp. 1:1–1:6.

[10] V. Cortellessa, A. Di Marco, C. Trubiani, An approach for modeling and detecting software performance antipatterns based on first-order logics, Journal of Software and Systems Modeling-DOI: 10.1007/s10270-012-0246-z, 2012.

[11] V. Cortellessa, A. Martens, R. Reussner, C. Trubiani, A Process to Effectively Identify "Guilty" Performance Antipatterns, in: Fundamental Approaches to Software Engineering, 2010, pp. 368–382.

[12] C. U. Smith, L. G. Williams, More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot, in: Computer Measurement Group Conference, 2003, pp. 717–725.

[13] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, C. Trubiani, Digging into UML models to remove performance antipatterns, in: ICSE Workshop Quovadis, 2010, pp. 9–16.

[14] UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group, Inc. (2005), *http://www.omg.org/cgi-bin/doc?formal/05-07-04*.

[15] OCL 2.0 Specification, OMG document formal/2006-05-01, Object Management Group, Inc. (2006), *http://www.omg.org/cgi-bin/doc?formal/06-05-01*.

[16] UML Profile for MARTE beta 2, OMG document ptc/08-06-09, Object Management Group, Inc. (2008), *http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf*.

[17] W. J. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, 1998.

[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[19] M. Meyer, Pattern-based Reengineering of Software Systems, in: Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, 2006, pp. 305–306.

[20] L. G. Williams, C. U. Smith, PASA(SM): An Architectural Approach to Fixing Software Performance Problems, in: International Computer Measurement Group Conference, 2002, pp. 307–320.

[21] V. Cortellessa, L. Frittella, A framework for automated generation of architectural feedback from software performance anal-

ysis, in: European Performance Engineering Workshop, 2007, pp. 171–185.

[22] T. Parsons, J. Murphy, Detecting Performance Antipatterns in Component Based Enterprise Systems, Journal of Object Technology 7 (3) (2008) 55–90.

[23] J. D. McGregor, F. Bachmann, L. Bass, P. Bianco, M. Klein, Using arche in the classroom: One experience, Tech. Rep. CMU/SEI-2007-TN-001, Software Engineering Institute, Carnegie Mellon University (2007).

[24] J. Xu, Rule-based Automatic Software Performance Diagnosis and Improvement, in: Workshop on Software and Performance, 2008, pp. 1–12.

[25] A. Sabetta, D. C. Petriu, V. Grassi, R. Mirandola, Abstraction-raising transformation for generating analysis models, in: MoDELS Satellite Events, 2005, pp. 217–226.

[26] A. Kavimandan, A. S. Gokhale, Applying Model Transformations to Optimizing Real-Time QoS Configurations in DRE Systems, in: International Conference on the Quality of Software Architectures (QoSA), 2009, pp. 18–35.

[27] E. Ipek, S. A. McKee, K. Singh, R. Caruana, B. R. de Supinski, M. Schulz, Efficient architectural design space exploration via predictive modeling, ACM Transactions on Architecture and Code Optimization (TACO) 4 (4) (2008) 1–34.

[28] A. Aleti, S. Björnander, L. Grunske, I. Meedeniya, ArcheOpterix: An extendable tool for architecture optimization of AADL models, in: ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), 2009, pp. 61–71.

[29] P. H. Feiler, D. P. Gluch, J. J. Hudak, The Architecture Analysis and Design Language (AADL): An Introduction, Tech. Rep. CMU/SEI-2006-TN-001, Software Engineering Institute, Carnegie Mellon University (2006).

[30] A. Martens, H. Koziolek, S. Becker, R. H. Reussner, Automatically improve software models for performance, reliability and cost using genetic algorithms, in: WOSP/SIPEW International Conference on Performance Engineering, 2010, pp. 105–116.

[31] A. Koziolek, H. Koziolek, R. Reussner, Peropteryx: automated application of tactics in multi-objective software architecture optimization, in: International ACM SIGSOFT Conference on the Quality of Software Architectures, 2011, pp. 33–42.

[32] K. Krogmann, M. Kuperberg, R. Reussner, Using genetic search for reverse engineering of parametric behavior models for performance prediction, IEEE Transactions on Software Engineering 36 (6) (2010) 865–877.

[33] L. Kapová, B. Buhnova, A. Martens, J. Happe, R. Reussner, State dependence in performance evaluation of component-based software systems, in: WOSP/SIPEW International Conference on Performance Engineering, 2010, pp. 37–48.

[34] C. U. Smith, L. G. Williams, New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot, in: Computer Measurement Group Conference, 2002, pp. 667–674.

[35] M. Hauck, M. Kuperberg, K. Krogmann, R. Reussner, Modelling layered component execution environments for performance prediction, in: International Symposium Component-Based Software Engineering, 2009, pp. 191–208.

[36] D. Arcelli, V. Cortellessa, C. Trubiani, Influence of numerical thresholds on model-based detection and refactoring of performance antipatterns, First Workshop on Patterns Promotion and Anti-patterns Prevention (2013).

[37] V. Cortellessa, M. De Sanctis, A. Di Marco, C. Trubiani, Enabling performance antipatterns to arise from an adl-based software architecture, in: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA, 2012, pp. 310–314.

[38] I. Malavolta, M. Di Marcello, F. Gallo, L. Iovino, S. Pace, Bus on Air, Business-Plan Competition (2010).
URL `http://bpc.univaq.it/index.php?id=1017`

[39] M. Autili, L. Berardinelli, D. Di Ruscio, C. Trubiani, Providing lightweight and adaptable service technology for information and communication (plastic) in the mobile ehealth case study, in: ICSE Workshop Pesos, 2012, pp. 69–70.

[40] SAE, Architecture Analysis and Design Language (AADL), June 2006, as5506/1, http://www.sae.org.