Detection and Solution of Software Performance Antipatterns in Palladio Architectural Models

Catia Trubiani
Università degli Studi dell'Aquila
67010 L'Aquila, Italy
catia.trubiani@univaq.it

Anne Koziolek
Karlsruhe Institute of Technology,
76131 Karlsruhe, Germany
martens@kit.edu

ABSTRACT

Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems. Performance Antipatterns document, from a performance perspective, common mistakes made during software development as well as their solutions. The definition of performance antipatterns concerns software properties that can include static, dynamic, and deployment aspects. Currently, such knowledge is only used by domain experts; the problem of automatically detecting and solving antipatterns within an architectural model had not yet been empirically addressed. In this paper we present an approach to automatically detect and solve software performance antipatterns within the Palladio architectural models: the detection of an antipattern provides a software performance feedback to designers, since it suggests the architectural alternatives to overcome specific performance problems. We implemented the approach and a case study is presented to demonstrate its validity. The system performance under study has been improved by 50% by applying antipatterns' solutions.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques, Performance Attributes; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.2.11 [**Software Engineering**]: Software Architectures.

General Terms

Performance, Antipatterns, Design.

Keywords

Software Performance Feedback, Performance Antipatterns, Palladio Component Model.

ICPE/WOSP/SIPEW'11, March 14–16, 2011, Karlsruhe, Germany.

1. INTRODUCTION

Software performance is a pervasive quality difficult to model, because it is affected by every aspect of the design and execution environment. The future trend in this domain is an automatic performance optimization of architecture, design and run-time configuration [26]: 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. Rounded boxes in the figure represent operational steps whereas square boxes represent input/output data. Dashed lines divide the process in three different phases: in the *modelling* phase, performance analysts build an (annotated[1]) software model; in the *analysis* phase, a performance model is obtained through model-to-model transformation, 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 are well-covered by several approaches [4] that introduce automation in all steps (e.g. [27, 6]). There is, however, a clear lack of automation in the refactoring phase, which shall improve the software architecture based on the analysis results. The goal of the refactoring phase, whose core step is the *result interpretation and feedback generation* step (see Figure 1), is to look for performance flaws 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.

A few approaches [11, 20, 28] address to automate the refactoring step. They, however, either do not implemented the proposed automation [11], or focus on technology-specific performance problems only [20], or address performance problems at the performance model level only [28], so that feedback to the design is not directly available.

Performance antipatterns [21] are descriptions of performance problems commonly encountered by performance en-

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

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

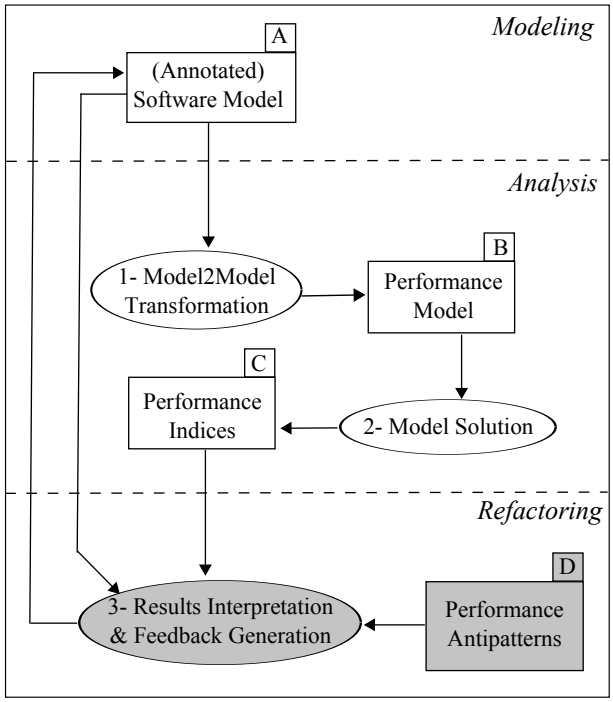

Figure 1: Software performance modelling and analysis process.

gineers in practice and represent a promising instrument 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 affects the performance indices negatively, thus to support the *results interpretation* activity; on the other hand, a performance antipattern definition includes a solution description that lets the designer devise refactoring actions, thus to support the *feedback generation* activity.

The main source of performance antipatterns [23] defined a number of 14 notation- and domain-independent antipatterns. Few other papers present technology-specific performance antipatterns.

In our work, step 3 in the refactoring phase takes the definition of performance antipatterns (label D) as an additional input. Performance antipatterns are detected in the (annotated) software model thus to address the *results interpretation*, and a refactored software model is built by solving the antipatterns, as suggested in their definition, thus to address the *feedback generation*.

The performance antipatterns we consider are not specific to any modelling language. In our previous work, [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 [9] we showed how performance antipatterns can be defined and detected in UML [3] models using OCL [1] queries, but we have not yet automated their solution.

In this paper we present the first automated approach to automatically detect and solve performance antipattern in a design-level software modelling language. We examined performance antipatterns within the Palladio Component Model (PCM) [5], which is a domain specific modelling language to describe component-based software architectures. Starting from the natural language description of antipatterns [23], we define a set of *rules* and *actions* expressed in terms of the PCM meta-model elements, in order to automate both the detection and the solution of performance antipatterns in PCM models.

Note that other software modelling languages can be considered, too, if the concepts for representing antipatterns are available; for example, architectural description languages such as AADL [2] can be also suited to apply the approach. As future work, we plan to investigate the representation of antipatterns in different modelling languages in order to gain experience for a more general framework, independent of any modelling notation.

In this paper the software process of Figure 1 is instantiated in Table 1[3]: the software system is modelled with PCM; the transformation from the software model to the performance model generates the simulation code for the PCM simulation tool SimuCom [5]; the performance model is then simulated to obtain the performance indices of interest: response time, utilisation, throughput, etc.

The contribution of this paper is represented by the two bottom most entries of Table 1. A set of *rules* and *actions* are defined to overcome the performance flaws: each rule characterizes the properties to detect performance antipatterns in the PCM model under analysis, and each action describes the changes to solve antipatterns in such model.

Using our approach, performance analysts can detect and solve performance problems more quickly. Instead of manually analysing the result indices of performance analyses and coming up with possible alternatives, they only have to assess the refactoring actions created by the application of our antipattern detection and solution approach.

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

Table 1: A customized overview of the process.

The rest of the paper is organized as follows: Section 2 compares the related works to our approach. An overview of which performance antipatterns are detectable or solvable within the PCM context is given in Section 3. Section

[3]Note that for input/output data we refer to modelling notations, numerical values, and queries, whereas for operational steps we refer to methodologies.

4 describes in detail some examples of the antipatterns that can be detected and solved within the PCM modelling notation: such antipatterns are represented as a set of rules for the identification of the problem, as well as a set of actions for the application of the solution. Section 5 reports the case study that motivates the beneficial effects of detecting and solving performance antipatterns. Assumptions, limitations, and open issues are discussed in Section 6, and finally Section 7 concludes the paper by giving a summary of the work and directions for future research.

2. RELATED WORK

The term *antipattern* appeared for the first time in [8] in contrast to the trend of focus on positive and constructive solutions. Differently from patterns [14], antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences. In general, antipatterns can be applied in different domains: for example, in [25] data-flow antipatterns help to discover errors in workflows; as another example, in [7] antipatterns help to discover multi-threading problems of java applications.

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. We are particularly interested in *technology independent* performance antipatterns [23], because our goal is to tackle the problem at the modelling level, by looking at the architectural design of software systems and localizing the most critical parts, from a performance perspective. *Technology specific* antipatterns have been specified in [13, 24] but they are out of our interest, because they focus on source code (e.g. an over use of session beans) and performance issues emerge only after the actual implementation of the system.

A first proposal of automated generation of feedback due to the software performance analysis driven by antipatterns can be found in [11], where the detection of performance flaws is demanded to the analysis of Layered Queued Network (LQN) models and uses informal interpretation matrices reasoning only on performance indices and violated requirements. This paper 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 [11], it provides support to automatically solve the detected antipatterns.

The issue of detecting performance antipatterns has been addressed in [20], where a rule-based 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.

Another interesting work on the software performance diagnosis and improvements has been proposed in [28]: rules to identify patterns of interaction between resources are defined. Performance flaws are identified before the implementation of the software system, however only the antipatterns 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 the LQN performance model, so that the translation of these model properties into design changes could (i) hide some possible refactoring solutions as well as (ii) be impossible due to design constraints. Our approach refers both to performance and design features of the software system in the feedback generation process in order to maintain the information we need to choose the best design alternatives.

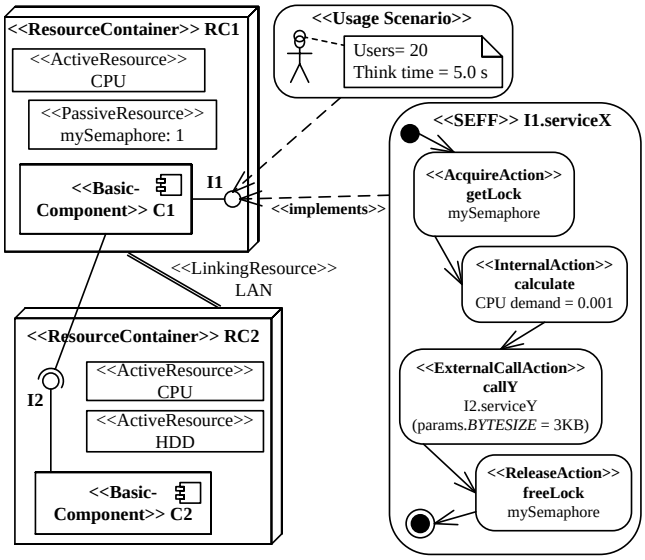

Figure 2: An example of a PCM model.

In [18], 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 uses random changes of the architecture instead of making use of performance knowledge. In this paper, we demonstrate the benefit of using antipatterns, since they allow to converge quickly towards performance improvements by suggesting suitable architectural alternatives.

3. OVERVIEW OF PERFORMANCE ANTIPATTERNS IN PCM

In this section we provide an overview of the performance antipatterns that can be specified in the Palladio Component Model (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, the simple example in Figure 2 contains 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 `typewriter font`. Note that only features relevant to this paper are shown here, other PCM features can be found in [5].

A software system in the PCM is modelled as a set of com-

ponent (here: Basic Components C1, C2). 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. Components are assembled to a System by connecting provided and required Interfaces. For example, the Interface I2 provided by component C2 satisfies C1's requirement of that Interface.

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 this example, a semaphore Passive Resource with capacity 1 is modelled in Resource Container RC1. Active Resources have 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 the components inside the container in this example. Resource Containers are connected by Linking Resources, whose timing behaviour is determined by the size of sent data.

Service Effect Specifications (SEFFs) describe the behaviour of the services offered by the Basic Component. A SEFF contains a sequence of actions. External Call Actions model calls to required interfaces. For example, serviceX of component C1 calls the 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, which is used to determine the linking resource load. Internal Actions specify 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 0.01 each time it is called. Acquire Actions and Release Actions model the use of Passive Resources in the PCM. Control flow structures are modelled with LoopActions, BranchActions, and ForkActions (not shown in the example).

3.2 Performance Antipatterns in the PCM

In general, in a modelling language, there are antipatterns that can be automatically detected and solved, others that can be automatically detected, but not automatically solved, and finally some others 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 [23], 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 the processing by judging the importance of the application code that it is an abstraction level not included in software models.

The list of the antipatterns we consider (see Table 2) have been enriched with an additional attribute: the *Single-value* antipatterns are detectable by mean, max or min values of performance indices, whereas the *Multiple-values* antipatterns are detectable by the trend or evolution of the performance indices along the time (see more details in [10]). 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, if it is automatically *detectable* and *solvable* in PCM models. The entries of Table 2 can be of three different types with the following meaning: $\surd$ (i.e. yes), $\times$ (i.e. no), and $-$ denotes that the corresponding operation does not make sense, i.e. if an antipattern cannot be detected it is obvious that it cannot be solved.

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

Table 2: Performance Antipatterns automatically detectable and solvable in PCM modelling language.

Table 2 points out that the most interesting antipatterns in the PCM context are: Concurrent Processing Systems, Extensive Processing, and One-Lane Bridge. Such antipatterns can be referred as *solvable* antipatterns, since they can be automatically detected and solved. More details on the detection and the solution of antipatterns are provided in Section 4.

Table 2 reveals that there are currently 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* antipatterns, since it is only possible to devise some actions to be manually performed (see Section 4.2).

Table 2 indicates that there are currently 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 [23]. In the PCM, data flow is more abstract and does not include information on data formats. However, it might be possi-

ble to replace the current modelling language to specify the behavioural description of services, i.e. the PCM service effect specification (SEFF), by another behavioural description language that includes such detail.

Excessive Dynamic Allocation is an antipattern whose bad practice is on unnecessarily creating and destroying objects during the execution of an application [21]. 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 [17], 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 on the increasing value of the response time and a decreasing throughput over time [23]. It might be detected by introducing the concept of *state* as suggested in [16], and it might be possible to inform the designer 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 in comparison of accomplishing the real work, because there are too many processes in comparison to the available resources [22]. Currently thrashing cannot be modelled in the PCM, but it might be added by introducing layered execution environment models, as suggested in [15].

4. ANTIPATTERNS-BASED PROCESS

In this section, we describe an antipatterns-based process to improve the performance of software architectural models. Figure 3 details the software performance modelling and analysis process of Figure 1: the refactoring phase is explicitly represented in two main operational steps: (i) *detecting antipatterns* provides the localization of the critical parts of software architectures thus to address the results interpretation problem; (ii) *solving antipatterns* suggests the changes to be applied to the software model under analysis, thus to address the feedback generation problem.

Figure 3 shows that several *iterations* can be conducted to find the software model that best fits the users requirements. Because several antipatterns may be detected in a software model, and additionally, several solution actions may be available for solving an antipattern, a set of candidates (e.g. $Candidate_1$, ..., $Candidate_h$ in the first iteration) are generated as a result of the refactoring step. Then, the detection and solution approach can be iteratively applied to all newly generated candidates, to further improve the system.

The activities of detecting and solving antipatterns are performed by reasoning on the natural language definition of the *problem* and *solution* specifications [23], as reported in Table 3. We selected some performance antipatterns from Table 2 and distinguish between *semi-solvable* and *solvable* in the leftmost column (see Section 3).

Table 3 lists the performance antipatterns we examine in this section. Hence, the performance antipattern specification is given in terms of *rules* representing the problem (see Section 4.1) and *actions* representing the solution (see Section 4.2). Both rules and actions are expressed in terms of PCM meta-model elements presented in Section 3.1.

Note that the rules and the actions we propose reflect our interpretation of the natural language; other researchers might interpret and formalize the antipatterns differently.

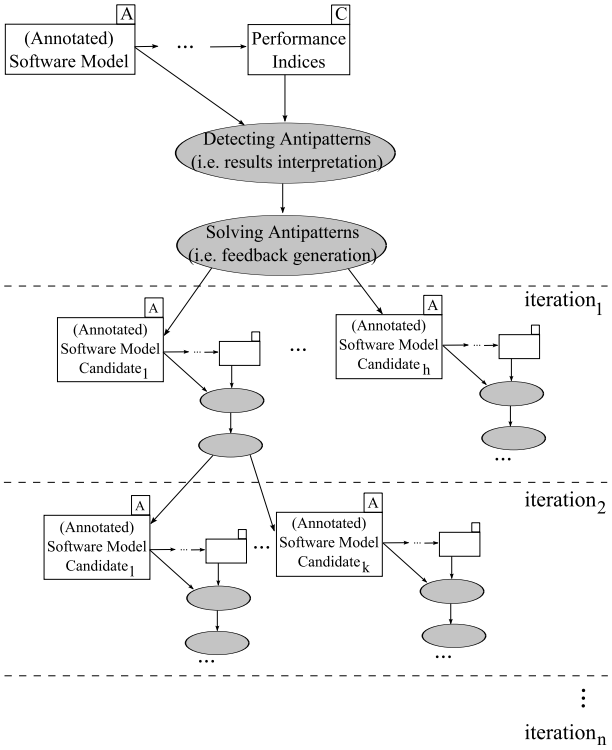

Figure 3: An antipatterns-based process for Software Performance Feedback.

This unavoidable gap is an open issue in this domain, and certainly requires a wider investigation to consolidate the formal definition of antipatterns.

Process afterthoughts are discussed in Section 4.3 where we propose a more compact graphical notation to summarize the process and we devise termination criteria.

4.1 Detecting antipatterns in PCM

From the informal representation of the *problem* (see Table 3) a set of *rules* is built, where each rule addresses part of the antipattern problem specification. An antipattern is detected if all its rules are fulfilled by a PCM model.

Note that the rules we propose are aimed at capturing bad practices hence 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 a critical point of the whole approach. Some sources can be used to perform this task such as: (i) the system requirements; (ii) the domain expert's knowledge; (iii) the evaluation of the system under analysis[4].

Blob is an antipattern whose problem is on the excessive message traffic generated by a single class or component [21]. The detection of the antipattern can be performed with the following rules.

[4]For example the upper bound for the hardware resource utilization can be estimated as the average utilization value overall the resources in the software system, plus the corresponding variance.

	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 processors.	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 3: Examples of performance antipatterns [23].

Usage Rule - a complex `Basic Component`, e.g. bc_x, depends on many other basic components, i.e. it requires many `Interface`s. It might means 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, i.e. its `External Call Action`s have a high frequency of being executed. It might mean that resources managing such communication could suffer from a performance perspective.

Utilisation Rule - if the bc_x basic component and the surrounding ones (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 `ActiveResource`s of rc_x; otherwise, if basic components are distributed on different `Resource Container`s, 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. This goal is performed by extracting the utilisation performance index from the simulation results.

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

Circuitous Treasure Hunt is an antipattern whose problem is on a inadequate organization of data that lead the system to look in several places to find the information it needs [23]. The detection of the antipattern can be performed with the following rules.

DBinteraction Rule - there are at least two PCM `Basic Component`s, e.g. bc_x and bc_y, such that: a) bc_x often calls bc_y, i.e. there is one or several `External Call Action`s in bc_x's `SEFF`s that call `Interface`s provided by bc_y and that together have a high frequency of being executed; and b) bc_y is a database, which is modelled by annotating bc_x with a custom mark model.

Utilisation Rule - similarly to the Blob antipattern (look at the rule with the same name) it is important to check the utilisation of the `Active Resource`s of the `Resource Container` on which the database basic component bc_y is deployed. Such a `Resource Container`, e.g. rc_y, is consider critical if the utilisation of any of its `Active Resource`s 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 Resource`s pr_{x_t} which have a type t, with $t \in \{CPU, HardDisk\}$. This rule matches if the maximum utilisation among all the hard disk(s) in rc_y is greater 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 on an excessive number of requests to perform a task [23]. The detection of the antipattern can be performed with the following rules.

Interaction Rule - similarly to the Blob antipattern (look at 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. its `External Call Action`s have a high frequency of being executed. It might means 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 means 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 Component`s as captured from the `Allocation`, and the communicating components are all deployed on the same `Resource Container`, without optimizing the interface.

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

Concurrent Processing Systems is an antipattern whose problem is on an unbalanced distribution of workload among the available processors [23]. The detection of the antipattern can be performed with the following rules.

QueueLength Rule - the system cannot make effective use of available processors: there is at least one `Active Resource` rc_{xt} 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 greater than a threshold value named $Th_{ql}(t)$ (e.g. $Th_{ql}(CPU)$ =50 requests and $Th_{ql}(HardDisk)$ =70 requests) for that resource type t.

Utilisation Rule - the `Active Resource` rc_{xt} is over utilised, i.e. it has a high utilisation. 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 type t named, for example, $Th_{maxUtil}(t)$ (e.g. $Th_{maxUtil}(CPU) = 80\%$ and $Th_{maxUtil}(HardDisk)$ =70%).

UnbalancedLoad Rule - processors are not used in a well-balanced way, there is at least another `Resource Container` instance, e.g. rc_y whose `Active Ressource`s of type t are less utilized in comparison to the ones of rc_x. The comparison of processing resources among the PCM containers rc_x and rc_y is performed by checking the utilisation according to their type t (i.e. CPU, HardDisk). Note that this goal is performed by extracting the utilisation performance index of the `Active Resource`s 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 type t named $Th_{minUtil}(t)$ (e.g. $Th_{minUtil}(CPU) = 30\%$ and $Th_{minUtil}(HardDisk) = 20\%$).

The output of the detection rules is a set of tupels with three elements: two `Resource Container`s satisfying the defined rules (e.g. rc_x and rc_y) and the critical resource type t (e.g. CPU, HardDisk).

Extensive Processing is an antipattern whose problem is on a not efficient management of requests: "lighter" requests are delayed, since they wait for "heavier" ones [22]. The detection of the antipattern can be performed with the following rules.

Structural Rule - there are two `SEFF`s, 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 from $seff_c$, either $seff_a$ of $seff_b$ is called, (ii) 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 PCM `Resource Container` hosting $seff_a$ and $seff_b$ that disables their concurrency.

ResourceDemand Rule - `SEFF` $seff_a$ has a high resource demand, i.e. its contained `Internal Action`s have a high resource demand. 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. the `SEFF` $seff_a$ 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 an heavy computation, i.e. the utilisation of one of its `Active Resource`s 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 on processes that are not allowed to be processed concurrently [21]. The detection of the antipattern 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. the queue length is greater 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).

4.2 Solving antipatterns in PCM

From the informal representation of the *solution* (see Table 3) a set of *actions* is built, where each action addresses part of the antipattern solution specification.

Blob, Circuitous Treasure Hunt, and Empty Semi Trucks are antipatterns whose solution is not automated because components are considered black-box elements in the PCM, and the component's internal behaviour cannot be restructured. However some actions can be suggested to and reviewed by the designer. Alternatively, the designer 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 from 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 communicating with the database and avoiding 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; such action can be performed by batching the messages, i.e. joining all small messages in few messages of a bigger size. The third action is aimed at optimizing the usage of the interface by reducing the remote communication;

such action can be performed by demanding 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 is 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 [23]. The solution of the antipattern can be performed with the following actions.

BalanceLoad Action - if the `Resource Container`s rc_x and rc_y[5] offer the same `Interface`s, change the scheduling algorithms and distribute in a balanced way (from rc_x to rc_y) the requests for such services by modifying the probability to be called.

Mirror Action - mirror the `Basic Component`s 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 Container`s. Consider the available `Active resource`s (i.e. cpu(s), hard disk(s)) of the `Resource Container`s.

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. Such action is rather simplistic, and we do not expect many performance improvements by applying it.

Redeploy Action - redeploy some `Basic Component`s 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.

The first option is aimed at redeploying components on the basis of their resource demand types. The redeployment of components can be performed by evaluating the resource container rc_y and deciding whenever it is better to redeploy CPU-critical (high computation demand) components and/or HardDisk-critical (high storage demand) ones.

The second option is aimed at redeploying components on the basis of the utilisation of processing resources belonging to the resource containers under analysis. The resource containers can be considered more or less critical if: a) there is at least one `Active Resource` of type $t*$ (i.e. CPU, HardDisk) providing a violation, i.e. $utilisation(pr_{x_{t*}})$ exceeds a threshold (e.g. $Th_{pr} = 85\%$), and ignore the others that do not provide any violation; b) all the `Active Resource`s of type $t*$ provide a violation.

The third option is aimed at redeploying components on the basis of their communication and trying to deploy components communicating each other possibly on the same `Resource Container`. Network links can considered more or less critical if: a) two components communicate through a `Linking Resource` providing a violation, i.e. $utilisation(nl_x)$ exceeds a threshold (e.g. $Th_{net} = 75\%$), but such components also use other network links that do not provide any violation; b) all the network links provide a violation.

Extensive Processing is an antipattern whose solution looks at scheduling requests according to their processing load and/or importance [22]. The solution of the antipattern can be performed with the following actions.

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

Note that the detection of the Extensive Processing antipattern involves two different reasons in its *structural rule* (see Section 4.1) that we recall in the following:

IncreaseCapacity Action - increase the capacity of `Passive Resource` locking 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 Component`s containing $seff_a$ or $seff_b$ thus 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 [21]. The solution of the antipattern can be performed with the following action.

IncreaseCapacity Action - increase the capacity of the `Passive Resource`s. A smarter methodology can be devised in order to optimize the capacity by evaluating the minimal multiplicity able to solve performance issues.

4.3 Process afterthoughts

The aim of this section is to discuss some afterthoughts about the antipatterns-based process. Figure 4 depicts the process we presented in Figure 3 in a graph-like way: each node joins the (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 *prediction value*s, since such requirements represent what end-users expect from the system for the target performance properties to be fulfilled. Such graphical notation gives an immediate overview on the software model that might best fit the end-users requirements.

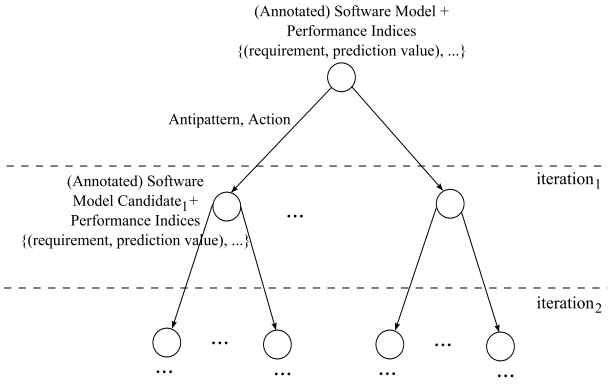

Figure 4: Process summary.

Note that we decided to separately consider each antipattern action (i.e. a single action gives rise to a software model candidate) since the automation of the refactoring is difficult for several reasons (see Section 6).

The number of software model candidates not necessarily increases between subsequent iterations, because each node is newly analysed and the number of detected antipatterns (and consequently their refactoring actions) can be fewer in comparison to the parent node.

Different termination criteria can be defined in 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 experimented; (iii) *#iterations* criterion, i.e. the process can be terminated if a certain number of iterations have been completed.

It is worth to notice that the solution of one or more antipatterns does not guarantee performance improvements in advance: the entire process is based on heuristics evaluations. The actions we propose build new PCM models, i.e. *candidate*s (see Figure 3), that replace the one under analysis, and the process can be newly iterated.

We have implemented the approach as an extension to the PCM Bench tool[6]. The current implementation can detect and solve three antipatterns in PCM models, namely Concurrent Processing Systems, Extensive processing and Onelane Bridge. The tool completely automates the described iterative search, supporting the stop criteria (ii) and (iii). We present results from experimentation with the extended tool in the next section.

5. CASE STUDY

In this Section we discuss a case study to demonstrate the validity of the antipatterns-based process, and it is organised as follows. First, Section 5.1 describes the PCM model of the system under analysis, the so-called business reporting system. Then, the stepwise application of the antipatterns-based process is performed, i.e. the detection of antipatterns (see Section 5.2) and their solution (see Section 5.3) across multiple iterations. Finally, Section 5.4 presents the experimental results we obtained.

5.1 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. To evaluate our approach, we intentionally introduced seven antipatterns in the system as described in Section 5.2.

Figure 5 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 make use of 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 5, 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.

[6]Both PCM Bench and extension can be downloaded at sdqweb.ipd.kit.edu/wiki/PerOpteryx, together with the case study model.

Not all services are inserted in the Figure 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 require an *InternalAction*, i.e. performed respectively by OnlineReporting and GraphicalReporting components, 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. For performance analysis, the software model is automatically transformed to simulation code, which is executed by the SimuCom simulation [5]. Additionally, the PCM usage model specifies 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.

The experimentation is conducted as follows. Starting from the BRS system modelled in PCM, our tool generates the simulation code and simulates the system with SimuCom. 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 antipatterns-based process is applied: if some performance antipatterns are detected in the model, their solution suggests the architectural alternatives that lead to obtain new software model candidates. Such models 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.

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 according to the defined usage model. The performance analysis of the BRS software model reveals that the response time of the system is 18.71 seconds (under a closed workload of 20 requests with thinking time of 5 seconds), so it does not meet the required 10 seconds. Since the requirement is not satisfied, we apply our approach to detect and solve performance antipatterns.

5.2 Detecting Antipatterns

Figure 5 shows some labels that indicate the detected antipatterns. All seven instances are found (PA_1, ..., PA_7), 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 solution.

Table 4 reports one example 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; the second column instantiates the *problem* by reasoning on the PCM model elements.

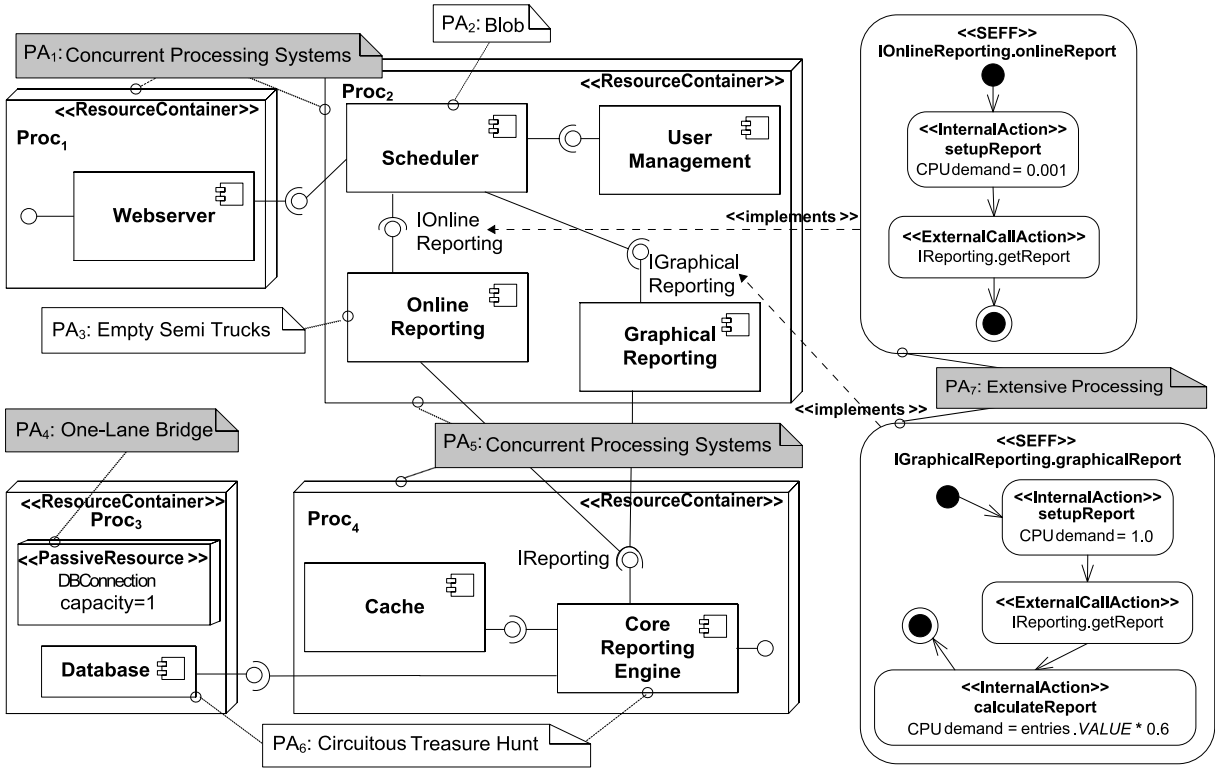

Figure 5: PCM Software Model for the BRS system.

Antipattern	Problem
Concurrent Processing Systems	*QueueLength Rule* - the PCM Active Resource CPU of $Proc_2$, not shown in Figure 5 for sake of readability, has a queueLength of 2.2 requests (i.e. greater than the threshold value, $Th_{ql}(CPU)$ =1.5 requests); *Utilisation Rule* - the PCM Active Resource CPU of $Proc_2$ has an utilisation of 62% (i.e. greater than the threshold value, $Th_{maxUtil}(CPU)$ =50%); *UnbalancedLoad Rule* - the PCM Active Resource CPU of $Proc_1$ has an utilisation of 6.9% (i.e. lower than the threshold value, $Th_{minUtil}(CPU)$ =10%).

Table 4: BRS- one example of detected antipatterns.

Note that several instances of the same antipattern type can be detected. For example, we found two instances of the Concurrent Processing Systems (not shown in Table 4 for sake of space) in the BRS system. Such antipatterns instances are not independent since they both contain the CPU of $Proc_2$ as the over utilised one. It is for this reason that we consider refactoring actions separately, to avoid infeasible architectural alternatives.

5.3 Solving Antipatterns

Table 5 lists some of the solved antipatterns for the BRS software model. In particular, two columns are defined: the first one indicates the *antipattern* type according to the Smith and Williams classification; the second one instantiates the *solution* by reasoning on the PCM model elements.

Applying a refactoring action from Table 5 results in a new software model candidate whose performance analysis reveals if the action is actually beneficial for the system under study. The solution of an antipattern, however, cannot guarantee performance improvements in advance because the whole process is based on heuristic rules.

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

Table 5: BRS- examples of solved antipatterns.

5.4 Experimental Results

Figures 6 reports our experimentation 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 antipatterns-based process are listed on the x-axis. Single points represent the response times observed after the separate solution of each performance antipattern.

Figure 6 summarizes the whole experimentation across the different *iterations*: 40 software model candidates are found, and the response time of the system spans from 18.71 sec-

onds (i.e. the initial value) to 9.26 sec (i.e. the value that fits with the requirement). Note that at each iteration a performance improvement is achieved up to the forth iteration, and the final improvement is roughly of 50%.

Figure 7 summarizes the process in the graph-like notation, as presented in Section 4.3: each node reports the performance index of our interest, i.e. RT(system), and its prediction value (e.g. 18.71 seconds in the root of the graph represents the prediction value for the initial system); each arc represents a refactoring *action* (e.g. the redeployment of the *GraphicalReporting* component from $Proc_2$ to $Proc_1$) applied to solve a detected *antipattern* (e.g. Concurrent Processing Systems). In our experimentation we applied the *fulfilment criterion* (see Section 4.3) to terminate the process, since the requirement is satisfied at the forth iteration and a software model candidate (i.e. the shaded node of Figure 7) able to cope with user needs is found.

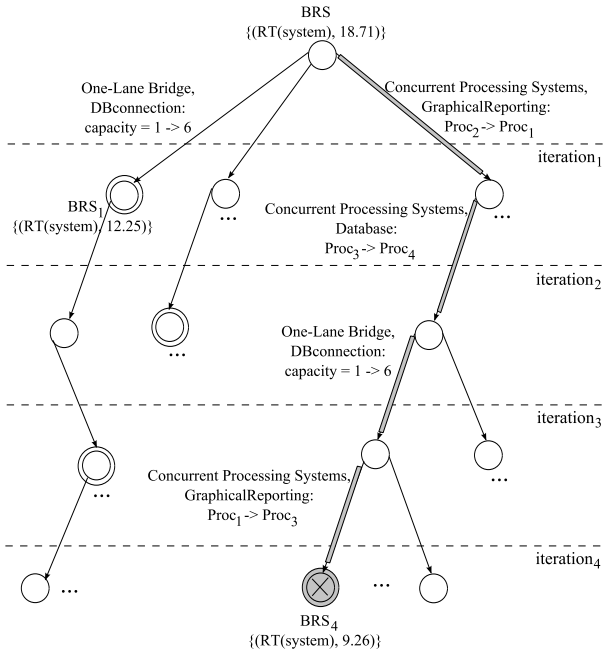

Figure 7: Process summary for the BRS system.

As shown in Figure 7, we can conclude that the software model candidate that best fits with user needs is obtained by applying the following refactoring actions: (i) the GraphicalReporting component is redeployed from $Proc_2$ to $Proc_1$; (ii) the Database component is redeployed from $Proc_3$ to $Proc_4$; (iii) the capacity of the passive resource DBconnection is increased from 1 to 6; (iv) the GraphicalReporting component is redeployed from $Proc_1$ to $Proc_3$.

6. DISCUSSION

The results we obtain by applying our approach of detecting and solving antipatterns are promising, although several open issues remain to be addressed 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.

The activity of solving antipatterns (i.e. the application of model refactoring actions on software models) implies some problems to be tackled. Once a number of performance antipatterns are detected, a certain strategy has to be introduced to decide which ones have to be solved in order to quickly converge towards an acceptable improvement of system performance. Such strategy has been partially discussed in [12], but several interesting issues have still to be faced, such as the simultaneous solution of multiple antipatterns. Note that the application of modifications is not commutative, because changing the order of solutions might change the performance results.

More experience could lead to refine the antipattern priorities on the basis of the application domain. For example the "Excessive Dynamic Allocation" antipattern might be of particular interest in object-oriented systems: although an automated solution of this antipattern is challenging, some tactics such as insertion or removal of caches might be experimented with.

Further issues in the solving antipatterns step might emerge for different factors: (i) architectural features (e.g. legacy constraints that do not allow to solve a certain antipattern, or incompatibility between solutions of different antipatterns); (ii) non-functional features (e.g. an antipattern solution is too expensive or it can badly affect the software dependability).

7. CONCLUSION

In this paper we presented an approach, based on antipatterns, that aims at identifying performance flaws in PCM models and removing them. The antipattern detection and solution activities are based on rules and actions, respectively, that formalize the informal existing definitions of performance antipatterns. We implemented the approach as an extension of the PCM Bench tool. In a case study, the approach was able to improve the system's performance under study by 50% by solving antipattern occurrences.

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 detection and solution approach.

As future work, we plan to combine antipattern detection and solution with multi-criteria evolutionary quality optimisation approaches such as [18]. 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.

8. ACKNOWLEDGMENTS

The authors would like to thank Vittorio Cortellessa and Heiko Koziolek, their comments are gratefully acknowledged. This work has been partially supported by VISION ERC project (ERC-240555).

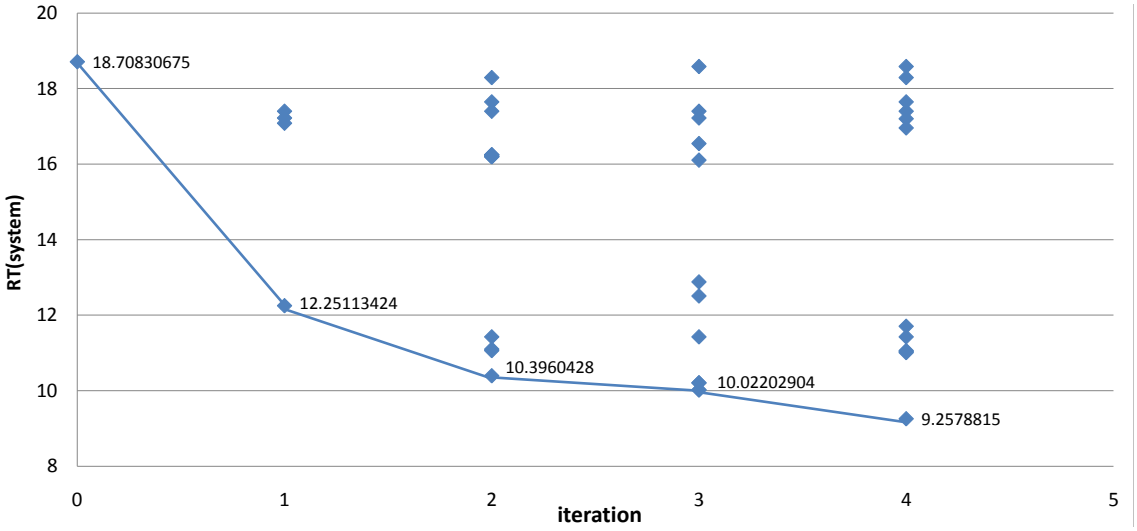

Figure 6: Response time of the *system* across the iterations of the antipattern-based process.

9. REFERENCES

[1] 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*.

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

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

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

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

[6] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable petrinet models. In *WOSP*, pages 35–45, 2002.

[7] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. In *Workshop on Dynamic Analysis*, pages 1–7, 2005.

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

[9] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani. Digging into UML models to remove performance antipatterns. In *ICSE Workshop Quovadis*, pages 9–16, 2010.

[10] V. Cortellessa, A. Di Marco, and C. Trubiani. Performance Antipatterns as Logical Predicates. In *IEEE International Conference on Engineering of Complex Computer Systems*, pages 146–156, 2010.

[11] V. Cortellessa and L. Frittella. A framework for automated generation of architectural feedback from software performance analysis. In *Proc. Formal Methods and Stochastic Models for Performance Evaluation*, 2007.

[12] V. Cortellessa, A. Martens, R. Reussner, and C. Trubiani. A Process to Effectively Identify "Guilty" Performance Antipatterns. In *Fundamental Approaches to Software Engineering*, pages 368–382, 2010.

[13] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf. *J2EE Antipatterns*. 2003.

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

[15] M. Hauck, M. Kuperberg, K. Krogmann, and R. Reussner. Modelling layered component execution environments for performance prediction. In *International Symposium Component-Based Software Engineering*, pages 191–208, 2009.

[16] L. Kapova, B. Zimmerova, A. Martens, J. Happe, and R. H. Reussner. State Dependence in Performance Evaluation of Component-Based Software Systems. In *WOSP/SIPEW*, pages 37–48, 2010.

[17] K. Krogmann, M. Kuperberg, and R. Reussner. Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. *IEEE Transactions on Software Engineering*, 2010. accepted for publication, to appear.

[18] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner. Automatically Improve Software Models for Performance, Reliability and Cost Using Genetic Algorithms. In *WOSP/SIPEW*, pages 105–116, 2010.

[19] M. Meyer. Pattern-based Reengineering of Software Systems. In *WCRE*, pages 305–306, 2006.

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

[21] C. U. Smith and L. G. Williams. Software Performance Antipatterns. In *WOSP*, pages 127–136, 2000.

[22] C. U. Smith and L. G. Williams. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. In *Computer Measurement Group Conference*, 2002.

[23] C. U. Smith and L. G. Williams. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *Computer Measurement Group Conference*, 2003.

[24] B. Tate, M. Clark, B. Lee, and P. Linskey. *Bitter EJB*. 2003.

[25] N. Trcka, W. M. van der Aalst, and N. Sidorova. Data-Flow Anti-Patterns: Discovering Dataflow Errors in Workflows. In *Conference on Advanced Information Systems*, volume 5565, pages 425–439. LNCS Springer, 2009.

[26] C. M. Woodside, M. Franks, and D. C. Petriu. The Future of Software Performance Engineering. In *Workshop on the Future of Software Engineering*, pages 171–187, 2007.

[27] C. M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by Unified Model Analysis (PUMA). In *WOSP*, pages 1–12, 2005.

[28] J. Xu. Rule-based Automatic Software Performance Diagnosis and Improvement. In *WOSP*, pages 1–12, 2008.