Electronic Communications of the EASST
Volume 59 (2013)

Special Issue of the First Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP 2013)

Experimenting the Influence of Numerical Thresholds on Model-based Detection and Refactoring of Performance Antipatterns

Davide Arcelli, Vittorio Cortellessa, Catia Trubiani

30 pages

Guest Editors: Aminata Sabané, Wei Wu
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Experimenting the Influence of Numerical Thresholds on Model-based Detection and Refactoring of Performance Antipatterns

Davide Arcelli[1], **Vittorio Cortellessa**[2], **Catia Trubiani**[3]

[1] davide.arcelli@univaq.it
[2] vittorio.cortellessa@univaq.it
[3] catia.trubiani@univaq.it
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica (DISIM)
Università degli Studi dell'Aquila
L'Aquila, Italy

Abstract: Performance antipatterns are well-known bad design practices that lead to software products suffering from poor performance. A number of performance antipatterns has been defined and classified and refactoring actions have also been suggested to remove them. In the last few years, we have dedicated some effort to the detection and refactoring of performance antipatterns in software models. A specific characteristic of performance antipatterns is that they contain numerical parameters that may represent thresholds referring to either performance indices (e.g., a device utilization) or design features (e.g., number of interface operations of a software component). In this paper, we analyze the influence of such thresholds on the capability of detecting and refactoring performance antipatterns. In particular, (i) we analyze how a set of detected antipatterns may change while varying the threshold values and (ii) we discuss the influence of thresholds on the complexity of refactoring actions. With the help of a leading example, we quantify the influence using precision and recall metrics.

Keywords: Software Performance Antipatterns, Detection, Refactoring, Numerical thresholds.

1 Introduction

In the software development domain, there is a high interest in the early validation of performance requirements because it avoids late and expensive fixes to consolidated software artifacts. Model-based approaches, pioneered under the name of Software Performance Engineering (SPE) by Smith [Smi07], aim at producing performance models early in the development cycle and using quantitative results from model solutions to refactor the design with the purpose of meeting performance requirements [WFP07].

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

useful to overcome performance problems (e.g., split a software component in two components and re-deploy one of them). The results interpretation is today exclusively based on the analysts' experience and therefore it suffers of lack of automation.

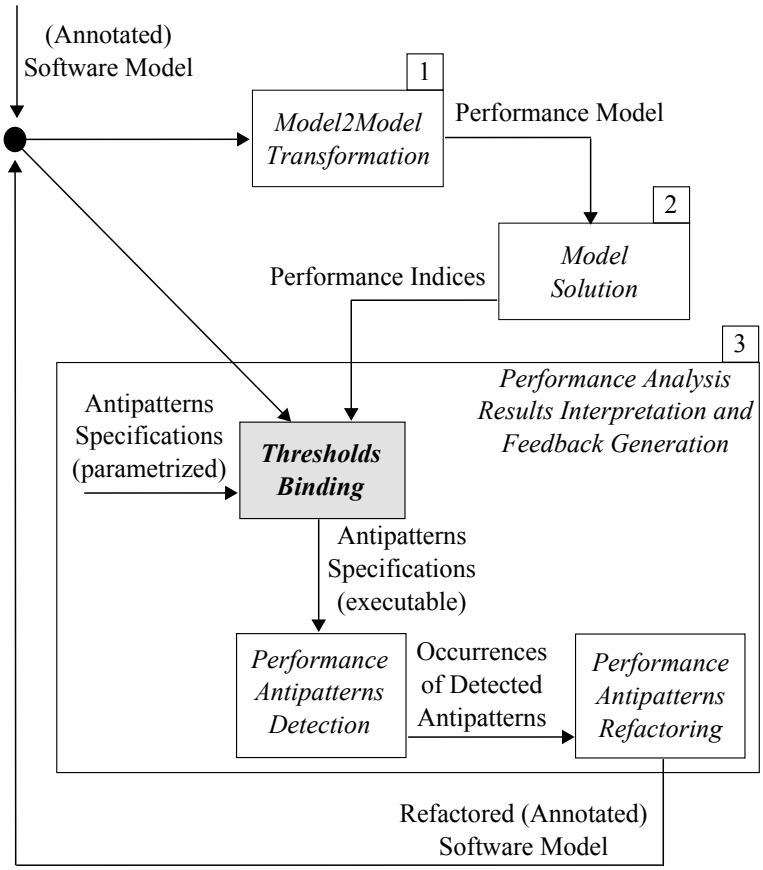


Figure 1: Model-based software performance analysis process.

Figure 1 illustrates a model-based software performance analysis process. It includes three main operational steps: (1) the *Model2Model Transformation* step takes as input an annotated[1] software model and generates a performance model [CMI11]; (2) the *Model Solution* step takes as input a performance model and produces a set of performance indices [LKGS84]; (3) the *Performance Analysis Results Interpretation and Feedback Generation* macro step takes as input both the software model and the performance indices to detect possible performance problems[2] and it provides a refactored (annotated) software model where problems have been removed. In particular, the refactored model is obtained with a semantics-preserving transformation that aims at improving the quality of the original software model. In other words, the functional aspects of this latter model remain unaltered after the transformation. For example, the interaction between

[1] Software model annotations support the performance analysis by specifying parameters like workload, resource demands, etc. [Obj09].

[2] A performance problem is an unfulfilled requirement, e.g., the estimated response time of a service is higher than the required one.

two components might be refactored to improve performance by sending fewer messages with larger data per message.

Few approaches have been recently introduced for this macro step [Xu12, MKBR10] (see more details in Section 2) as have been working on the detection and refactoring of performance antipatterns [CDE+10, TK11, CDDT12]. *Performance antipatterns* [SW03] are well-known bad design practices that lead to software products suffering from poor performance and they include solutions in terms of refactoring actions.

The macro step of Figure 1 has been detailed with the two main steps that we have envisaged in our approach, which are performance antipatterns detection and refactoring. A further preliminary step has been made explicit in Figure 1, i.e., the *thresholds binding* and it represents the focus of this paper.

In fact, a specific characteristic of performance antipatterns is that they contain numerical parameters that represent thresholds referring to either performance indices (e.g., *high*, *low* device utilization) or design features (e.g., *many* interface operations, *excessive* message traffic). The *thresholds binding* step takes as input parametrized antipatterns specifications, determines the numerical values for antipattern thresholds and gives as output executable antipatterns specifications[3].

The goal of this paper is to analyze the influence of such thresholds on the capability of detecting and refactoring performance antipatterns. In particular, (i) we analyze how a set of detected antipatterns may change while varying the threshold values and (ii) we discuss the influence of thresholds on the complexity of refactoring actions. The thresholds influence is also quantified with precision and recall metrics.

To complete the description of Figure 1, we remark that the whole process may be iterated several times to find the model that best fits the performance requirements. In fact, several antipattern occurrences may be detected in a software model, and several refactoring actions may be available for solving each antipattern. Hence, at each iteration the antipattern-based refactoring actions are aimed at building new (refactored) software models that undergo the same process.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 provides some background on the thresholds introduced for the specification of performance antipatterns. Section 4 presents an illustrative example where performance antipatterns have been detected and refactored while considering fixed thresholds. Section 5 provides a sensitivity analysis on the values of thresholds. Section 6 discusses the emerging issues derived from our experimentation. Section 7 concludes the paper by outlining the most challenging research topics in this area.

2 Related work

Several approaches have been recently introduced to specify and detect code smells and antipatterns [MPN+12, KPGA12, RRPK12, YM12, PZ12]. They range from manual approaches, based on inspection techniques [TSFB99], to metric-based heuristics [Mar04, OKAG10], using

[3] "Executable" means that these resulting specifications can be used in the detection step to browse the software model.

rules and thresholds on various metrics [MGDM10] or Bayesian Belief Networks [KVGS11]. In complement, our approach intends to work at the design level and it can be applied early in the software life-cycle.

A small number of methodologies [MGDM10, KVGS11] were introduced to formalize the concepts and properties of code smells at a high-level of abstraction, whereas no formalization had been yet provided for the performance antipatterns specified in [SW03].

Very few model-based approaches for the macro step of Figure 1 exist and, in [AC13], we compared the approaches working either on software or performance sides. On the software side, in [MKBR10], meta-heuristic search techniques are used for improving different non-functional properties of component-based software systems: evolutionary algorithms search the architectural design space for optimal trade-offs. The main limitation of such approach is that it is quite time-consuming because the design space may be huge. On the performance side, in [Xu12], performance problems are identified before the implementation of a software system, but they are based only on bottlenecks (e.g., the "One-Lane Bridge" antipattern) and long paths. The main limitation of such approach is that it only applies to Layered Queueing Network performance models, hence its portability to other notations is yet to be proven and it may be quite complex.

In the area of software design quality improvement, several search-based refactoring techniques have been proposed. In [SSB06], a search-based approach for refactoring the class structure of a software system is proposed, but it is limited to a restricted set of refactorings. In [HT07], search-based techniques are used to automatically discover useful refactorings aimed at improving the quality of software systems. Authors use the concept of Pareto optimality to search-based refactoring, hence multiple fitness functions lead to different Pareto optimal refactorings. In [OC08], multiple weighted metrics are combined into a single fitness function that is based on well-known measures of coupling between components. All these search-based approaches share the same limitation, i.e., the search space may be huge, so the search process may be time-consuming.

3 Thresholds in performance antipattern specification/representation

Performance antipatterns have been originally defined in natural language [SW03]. Hence, we first tackled the problem of providing a more formal representation by introducing first-order logic rules that express a set of system properties under which an antipattern occurs [CDT12]. More recently, we undertook the problem of removing performance antipatterns detected in software models by introducing a role-based approach that allows to formalize the refactoring embedded into performance antipattern definitions [ACT12].

Our formalization of antipatterns [CDT12] reflects our interpretation of their informal textual definitions. Other feasible interpretations of antipatterns can be provided. This unavoidable gap, that recurs in any formalization task, requires a wider investigation to consolidate the antipatterns definitions and is left for future work.

Performance antipatterns are very complex (as compared to other software patterns) because they are founded on different characteristics of software systems, spanning from static through

behavioral to deployment. They additionally include parameters related to design features and performance indices. In fact, antipattern parameters are related to design characteristics (e.g., *many* usage dependencies, *excessive* message traffic) and–or to performance results (e.g., *high*, *low* utilization), hence thresholds must be introduced.

Because we cannot avoid thresholds in antipatterns definition, detection and refactoring activities are heavily affected by the multiplicity and the estimation accuracy of thresholds an antipattern requires.

Table 1 contains a list of performance antipatterns [SW03]. Each row represents a specific antipattern that is characterized by four attributes: antipattern type, name, and number of design/performance thresholds. We have partitioned antipatterns in two different types [CDT12]: the ones detectable by single values of performance indices (such as mean, max, or min values), named *Single-value* performance antipatterns, and the ones requiring the trend (or evolution) of performance indices in time, named *Multiple-values* performance antipatterns. Due to these characteristics, performance indices needed to detect the latter type of antipatterns must be obtained via simulation or monitoring.

Table 1: Overview of antipatterns thresholds.

Antipatterns		Thresholds	
Types	Names	Design	Performance
Single-value	Blob	2	2
	Extensive Processing	2	2
	Empty Semi Trucks	2	1
	Excessive Dynamic Allocation	2	1
	"Pipe and Filter" Architectures	1	2
	Circuitous Treasure Hunt	1	1
	Tower of Babel	1	1
	Concurrent Processing Systems	0	5
	One-Lane Bridge	0	1
Multiple-values	The Ramp	0	2
	Traffic Jam	0	1
	More is Less	0	0

From Table 1, we observe that: (i) some antipatterns include both design and performance thresholds such as Blob, Extensive Processing, etc.; (ii) some antipatterns only include performance thresholds, such as Concurrent Processing Systems, One-Lane Bridge, etc.; (iii) there is one antipattern (i.e., the More is Less) without thresholds because it relies on configuration parameters (database and web connections, etc.) that are detected by run-time software analysis.

The binding of thresholds to concrete numerical values (e.g., *0.8* may denote high utilization for a hardware resource) is a crucial point of any formalization, because it affects the number of false positives and true negatives that may occur.

Different sources of information can be used to support the binding of thresholds, such as: (i) the system requirements; (ii) the domain experts knowledge; (iii) the evaluation of the system under analysis. In our previous work [CDT12], we provided some heuristics to calculate these thresholds.

In the following, we present examples of the Blob, Concurrent Processing Systems (CPS), and Traffic Jam (TJ) performance antipatterns [SW03], i.e., shaded entries of Table 1.

Blob - A Blob occurs when a component requires a *lot* of information from other ones, it generates *excessive* message traffic that lead to *over utilize* the device on which it is deployed or the network involved in the communication. The logic-based formula of the Blob antipattern has been defined in [CDT12] and reported in Equation (1), where swℍ and 𝕊 represent the set of all software components and services, respectively.

$$\begin{aligned} &\exists swE_x, swE_y \in \text{swℍ}, S \in \mathbb{S} \mid \\ &(F_{numClientConnects}(swE_x) \geq Th_{maxConnects} \\ &\vee F_{numSupplierConnects}(swE_x) \geq Th_{maxConnects}) \\ &\wedge (F_{numMsgs}(swE_x, swE_y, S) \geq Th_{maxMsgs} \\ &\vee F_{numMsgs}(swE_y, swE_x, S) \geq Th_{maxMsgs}) \\ &\wedge (F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \\ &\vee F_{maxNetUtil}(P_{swE_x}, P_{swE_y}) \geq Th_{maxNetUtil}) \end{aligned} \tag{1}$$

Table 2 reports the functions involved in the Blob specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 2: Functions specification for the Blob antipattern.

Functions	Descriptions
int $F_{numClientConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships where swE_x assumes the client role.
int $F_{numSupplierConnects}$ (SoftwareEntityInstance swE_x)	It counts the multiplicity of the relationships where swE_x assumes the supplier role.
int $F_{numMsgs}$ (SoftwareEntityInstance swE_x, SoftwareEntityInstance swE_y, Service S)	It counts the number of messages sent from swE_x to swE_y in a service S.
float $F_{maxHwUtil}$ (ProcessNode pn_x, type T)	It provides the maximum utilization among the hardware devices of a certain type $T = \{$cpu, disk, all$\}$ hosted by the processing node pn_x.
float $F_{maxNetUtil}$ (ProcessNode pn_x, ProcessNode pn_y)	It provides the maximum utilization among the network links joining the processing nodes pn_x and pn_y.

Table 3 reports the thresholds involved in the Blob specification [CDT12]: two thresholds ($Th_{maxConnects}$, $Th_{maxMsgs}$) refer to design features, whereas the other ones ($Th_{maxHwUtil}$, $Th_{maxNetUtil}$) are related to performance indices.

Heuristics for Blob thresholds can be defined as follows [CDT12]. $Th_{maxConnects}$ can be estimated as the average number of connections per component, by considering the entire set of software components in the software system, plus the corresponding variance. In a similar way, $Th_{maxMsgs}$ can be estimated as the average number of sent messages per software component, plus the corresponding variance. $Th_{maxHwUtil}$ can be estimated as the average value of utilization per hardware device, plus the corresponding variance. Similarly, $Th_{maxNetUtil}$ can be estimated as the average value of utilization per network link, plus the corresponding variance.

Table 3: Thresholds specification for the Blob antipattern.

	Thresholds	Descriptions
Design	$Th_{maxConnects}$	Maximum bound for the number of connections in which a component is involved
	$Th_{maxMsgs}$	Maximum bound for the number of messages sent by a component in a service
Performance	$Th_{maxHwUtil}$	Maximum bound for the hardware device utilization
	$Th_{maxNetUtil}$	Maximum bound for the network link utilization

CPS - A CPS occurs when processes cannot make effective use of available hardware nodes because of a non-balanced assignment of tasks to devices. Some hardware nodes are *over-utilized* and some others are *under-utilized*. CPS occurrences are denoted with ($hwNode_1$, $hwNode_2$), where $hwNode_1$ is the over-utilized hardware node and $hwNode_2$ is the under-utilized one. The logic-based formula of the CPS antipattern has been defined in [CDT12] and reported in Equation (2), where $\mathbb{P}$ represents the set of all the hardware nodes.

$$\begin{aligned} &\exists P_x, P_y \in \mathbb{P} \mid \\ &F_{maxQL}(P_x) \geq Th_{maxQL} \\ &\wedge[(F_{maxHwUtil}(P_x, cpu) \geq Th_{maxCpuUtil} \\ &\wedge F_{maxHwUtil}(P_y, cpu) < Th_{minCpuUtil}) \\ &\vee(F_{maxHwUtil}(P_x, disk) \geq Th_{maxDiskUtil} \\ &\wedge(F_{maxHwUtil}(P_y, disk) < Th_{minDiskUtil}))] \end{aligned} \tag{2}$$

Table 4 reports the functions involved in the CPS specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 4: Functions specification for the CPS antipattern.

Functions	Descriptions
float F_{maxQL} (ProcessNode pn_x)	It provides the maximum queue length among the hardware devices hosted by the processing node pn_x.
int $F_{maxHwUtil}$ (ProcessNode pn_x, type T)	It provides the maximum utilization among the hardware devices of a certain type $T = \{$cpu, disk, all$\}$ hosted by the processing node pn_x.

Table 5 reports the thresholds involved in the CPS specification [CDT12]: all the five thresholds (Th_{maxQL}, $Th_{maxCpuUtil}$, $Th_{minCpuUtil}$, $Th_{maxDiskUtil}$, and $Th_{minDiskUtil}$) are related to performance indices.

Heuristics for CPS thresholds can be defined as follows [CDT12]. Th_{maxQL} can be estimated as the average value of queue length per hardware device, plus the corresponding variance. $Th_{maxCpuUtil}$ can be estimated as the average value of utilization per processing device, plus the corresponding variance; in a similar way, $Th_{minCpuUtil}$ can be estimated as the average value

Table 5: Thresholds specification for the Concurrent Processing Systems antipattern.

	Thresholds	Descriptions
Performance	Th_{maxQL}	Maximum bound for the hardware device queue length
	$Th_{maxCpuUtil}$	Maximum bound for the processing device utilization
	$Th_{minCpuUtil}$	Minimum bound for the processing device utilization
	$Th_{maxDiskUtil}$	Maximum bound for the disk device utilization
	$Th_{maxDiskUtil}$	Maximum bound for the disk device utilization

of utilization per processing device, minus the corresponding variance. $Th_{maxDiskUtil}$ can be estimated as the average value of utilization per disk device, plus the corresponding variance; similarly, $Th_{minDiskUtil}$ can be estimated as the average value of utilization per disk device, minus the corresponding variance.

TJ - A TJ occurs when one problem causes a backlog of jobs that results in a wide variability in response time, which persists long after the problem has disappeared. The logic-based formula of the TJ antipattern has been defined in [CDT12] and reported in Equation (3), where $\mathbb{O}$ represents the set of all operation instances.

$$\begin{aligned} & \exists OpI \in \mathbb{O} \mid \\ & \frac{\sum_{1\leq t\leq k}|(F_{RT}(OpI,t)-F_{RT}(OpI,t-1))|}{k-1} < Th_{OpRtVar} \\ & \wedge |F_{RT}(OpI,k)-F_{RT}(OpI,k-1)| > Th_{OpRtVar} \\ & \wedge \frac{\sum_{k\leq t\leq n}|(F_{RT}(OpI,t)-F_{RT}(OpI,t-1))|}{n-k} < Th_{OpRtVar} \end{aligned} \tag{3}$$

Table 6 reports the functions involved in the TJ specification. The first column of the Table shows the function signatures and the second column provides their descriptions.

Table 6: Functions specification for the TJ antipattern.

Functions	Descriptions
float F_{RT} (OperationInstance OpI, timeInterval t)	It provides the estimated response time of the operation instance OpI at the time interval t

Table 7 reports the threshold involved in the TJ specification [CDT12]: $Th_{OpRtVar}$ is related to performance indices.

The binding of some thresholds is intrinsically more difficult than others. For example, both the Traffic Jam and The Ramp antipatterns refer to thresholds representing the maximum feasible slope of the response time (or throughput) observed in consecutive time slots. Such values are not easy to bind. Adaptive heuristics can be introduced to iteratively obtain more accurate threshold boundaries. For example, in case of The Ramp and Traffic Jam antipatterns, such heuristics may

Table 7: Threshold specification for the Traffic Jam antipattern.

	Thresholds	Descriptions
Performance	$Th_{OpRtVar}$	Maximum bound for the variability in response times of operations across simulation intervals

exploit historical data (obtained by previous performance analysis) to accurately tune the slope used as boundary for the increase of response time and the decrease of throughput.

4 Detection and refactoring of antipatterns: fixed thresholds

In this section, we present the example leading our experimentation. In particular, we first describe the (Annotated) E-Commerce System (ECS) software model and the performance indices obtained from its analysis. We then perform a preliminary antipatterns detection/refatoring step with fixed thresholds values.

4.1 An illustrative Example: ECS model

ECS is a web-based system that manages business data related to books and movies. Figure 2 shows the UML [Obj05] Use Case Diagram: a *Guest* may invoke the *BrowseCatalog* service, whereas a *Customer* may invoke two services, i.e., *Login* and *MakePurchase*.

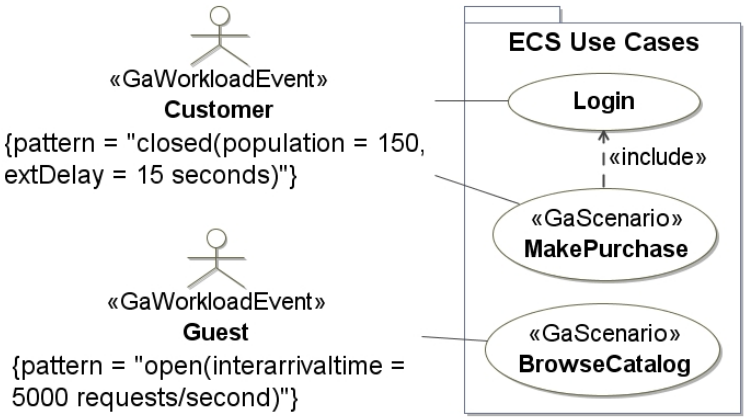


Figure 2: Use Case Diagram.

Software model annotations support performance analysis. In fact, in Figure 2, we observe that MARTE [Obj09] annotations have been added to specify the system workload. In particular, (i) a closed workload has been defined for the *MakePurchase* service, for which the number of users is set to 150 with an average thinking time of 15 seconds and (ii) an open workload has been defined for the *BrowseCatalog* service, for which the average arrival rate is set to 5,000 requests per second.

We assume to have a multi-view annotated software model, composed by *Static*, *Dynamic* and *Deployment Views*. Many modeling languages today (such as UML [Obj05], Palladio [BKR09],

Æmilia [BDC02]) allow to model different views of a software system to achieve a clear separation of concerns. For software analysis, a (possibly restricted) set of views is usually considered, where each view is properly annotated with parameters involved in the analysis process. The approaches in [BDIS04] represent a broad summary on the usage of multiple views in performance analysis.

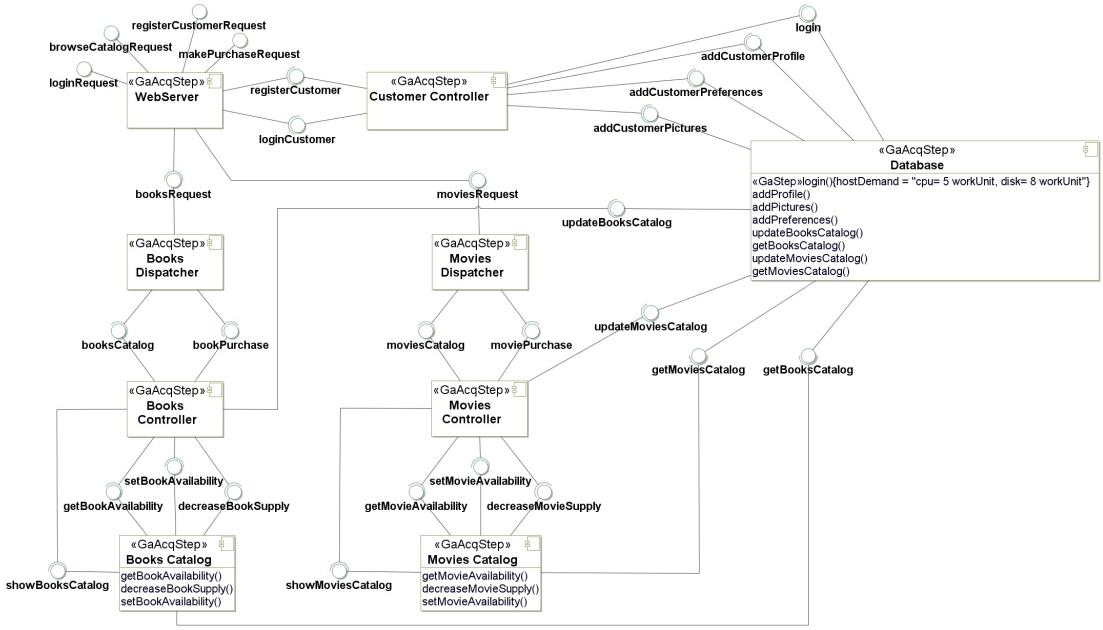
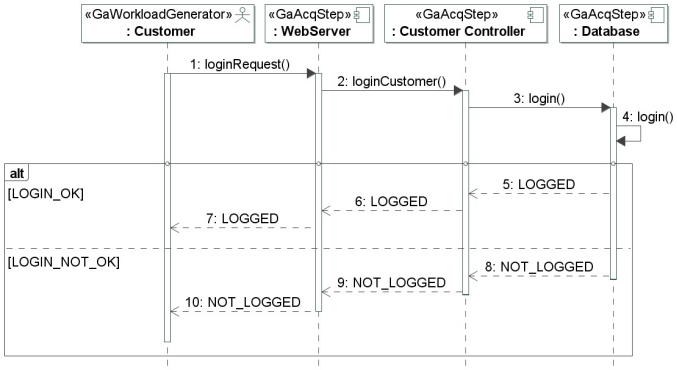


Figure 3: (Annotated) Static View.

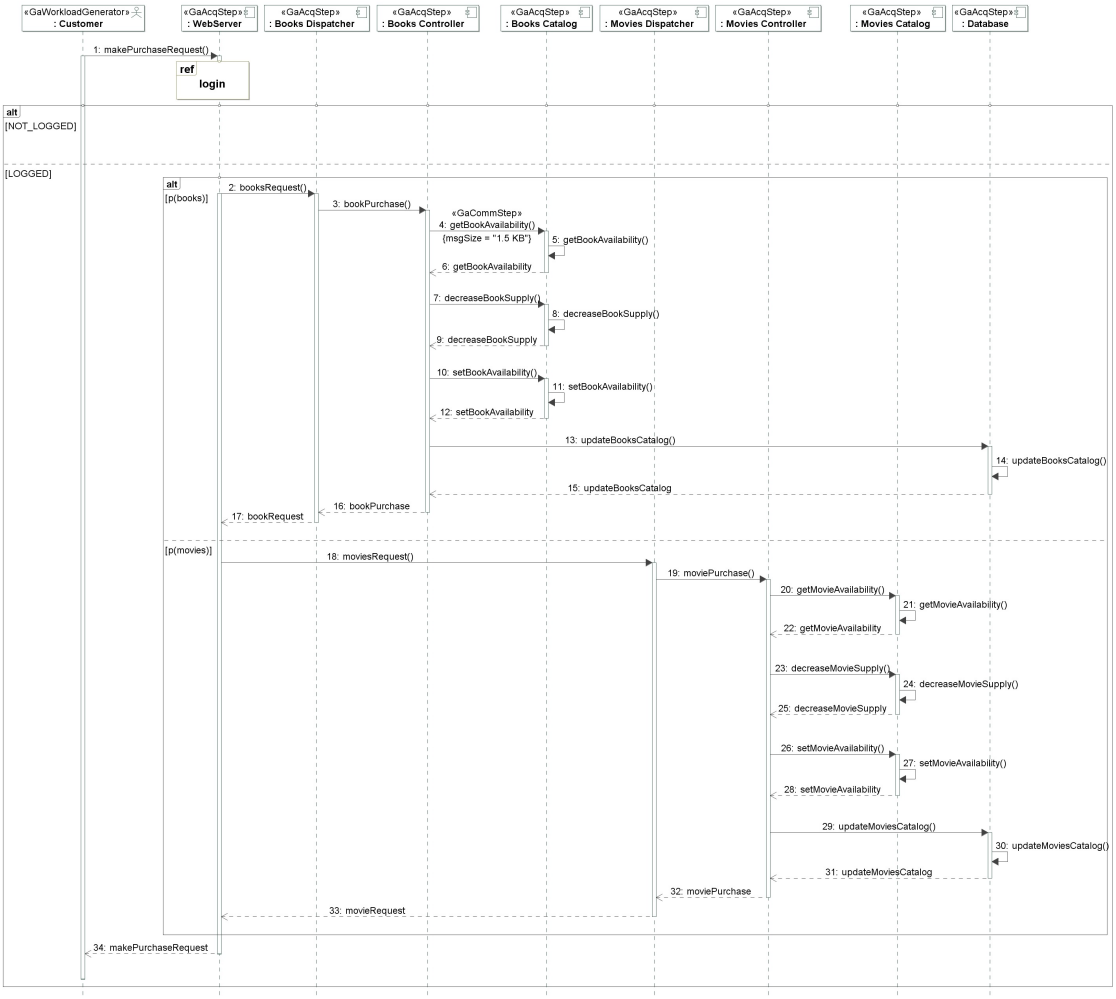
Several software components have been defined and connected in the *Static View* (see the UML Component Diagram in Figure 3). MARTE annotations have been added to specify the host demand required by components operations. For example, in Figure 3, we observe that the *login* operation provided by the *Database* component requires 5 and 8 work units for cpu and disk devices, respectively.

Among all system services, we focus here on: (i) *MakePurchase*, which is triggered whenever a customer wants to purchase a book or a movie, after authentication (see the UML Sequence Diagram in Figure 4); (ii) *BrowseCatalog*, which is triggered whenever a guest wants to browse book or movie catalogues (see the UML Sequence Diagram in Figure 5). MARTE annotations have been added to specify the message size of components communications. For example, in Figure 4, we observe that the *BooksController* communicates with the *BooksCatalog* by means of a message (*getBookAvailability*) whose size is equal to 1.5 KB.

The *Deployment View* (see the UML Deployment Diagram in Figure 6) shows the ECS allocation of software components on hardware nodes. Service requests from customers and guests (client-side) pass through the *Internet* and all the nodes in the server-side are connected by means of a 100 Mb/s *LAN* (as specified by MARTE annotations). Finally, for the sake of simplicity, we assume that both the client-side and the server-side are equipped with nodes having one central processing unit (cpu) and one disk and that all the processing and disk devices have the same characteristics.



(a) (Annotated) Dynamic View - *Login*



(b) (Annotated) Dynamic View - *MakePurchase*

Figure 4: (Annotated) Dynamic View: *Login* and *MakePurchase* services.

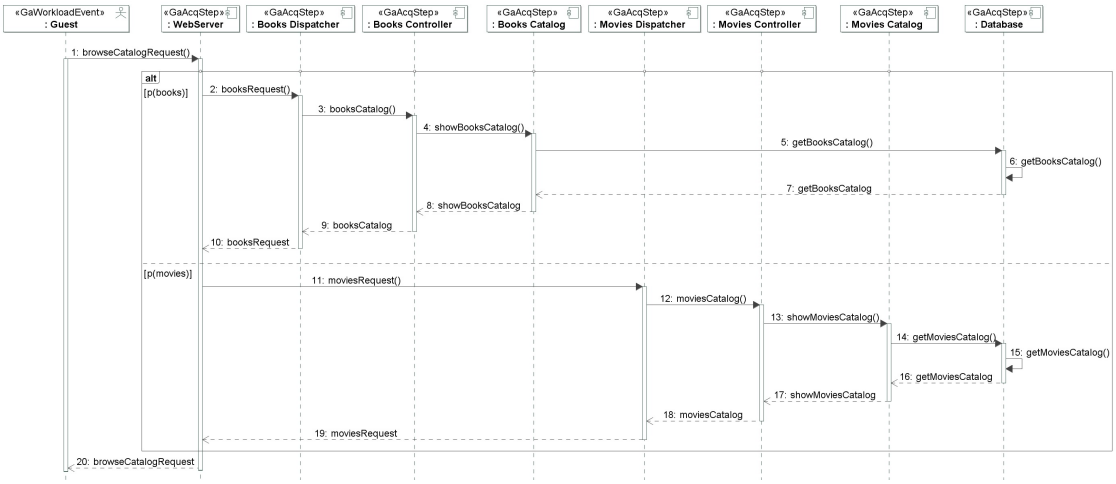


Figure 5: (Annotated) Dynamic View: *BrowseCatalog* service.

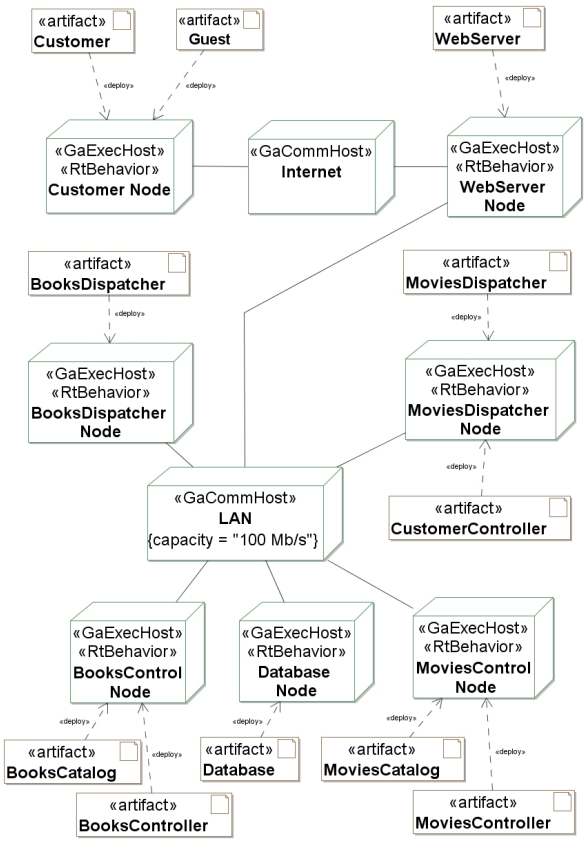


Figure 6: (Annotated) Deployment View.

We assume that performance requirements have been defined on the *MakePurchase* and *BrowseCatalog* services: (i) the average response time of the *MakePurchase* service must not exceed 1.83 seconds; (ii) the average response time of the *BrowseCatalog* service must not exceed 5 seconds. Both requirements must be fulfilled under a workload of 150 customers.

A.I Performance Annotations

Table 8 reports the average service demands (expressed in seconds) for the *MakePurchase* service. We have supposed an average thinking time of 15 seconds (see Figure 2) for each customer.

Table 8: *MakePurchase* service demands.

Node	D(*MakePurchase*) [sec]
CustomerNode	15
WebServerNode	0.015
BooksDispatcherNode	0.008
MoviesDispatcherNode	0.062
BooksControlNode	0.1
MoviesControlNode	0.105
DatabaseNode	0.09

Table 9 reports the average service demands (expressed in seconds) for the *BrowseCatalog* service. We have supposed an average arrival rate of 5,000 requests/second (see Figure 2).

Table 9: *BrowseCatalog* service demands.

Node	D(*BrowseCatalog*) [sec]
WebServerNode	0.06
BooksDispatcherNode	0.032
MoviesDispatcherNode	0.248
BooksControlNode	0.4
MoviesControlNode	0.42
DatabaseNode	0.36

A.II Performance Analysis

The performance analysis has been conducted by transforming the software model into a Queueing Network (QN) model [CM02] and by solving the latter with two techniques [Jai91]: (i) Mean Value Analysis for Single-value antipatterns and (ii) Simulation for Multiple-value antipatterns. Both solution techniques are supported by Java Modeling Tools (JMT) [CS11].

Table 10 shows the resulting performance indices for the ECS software model. In particular, the average response times (RT), utilizations (U), and queue length (QL) of hardware nodes are reported as well as the average response times of *MakePurchase* and *BrowseCatalog* services. The utilization of a hardware node is estimated as the maximum value overall of its cpu and disks devices [CDT12]. Hence, the column *DEVICE TYPE* of Table 10 contains *cpu* or *disk* whether the node utilization is determined by its cpu or disk, respectively.

As illustrated in Table 10, the considered requirements are violated because, under a workload of 150 users purchasing a product and browsing catalogues, the mean time elapsed in the server-

Table 10: Initial performance analysis results for the *MakePurchase* and *BrowseCatalog* services.

	RT [sec]	U [%]	QL [customers]	DEVICE TYPE
MakePurchase	17.16	-	150	-
BrowseCatalog	14.37	-	150	-
CustomerNode	15	-	131.1	-
WebServerNode	0.017	13.11	0.15	cpu
BooksDispatcherNode	0.009	6.99	0.07	cpu
MoviesDispatcherNode	0.134	54.19	1.17	cpu
BooksControlNode	0.672	87.4	5.88	cpu
MoviesControlNode	0.934	91.77	8.16	cpu
DatabaseNode	0.396	78.66	3.46	disk

side for each request (i.e., the average response time at the server-side) is larger than the stated requirement. In particular, the *MakePurchase* service has an average response time of 17.16 - 15 = 2.16 seconds, which is larger than the defined requirement of 1.83 seconds; the *BrowseCatalog* service has an average response time of 14.37 - 5 = 9.37 seconds, which is larger than the defined requirement of 5 seconds.

B. Antipatterns Detection and Refactoring

Before performing the preliminary antipatterns detection/solution step, we define the following refactoring actions used in our experimentation, which build upon our previous work on performance antipatterns removal [ACT12]:

- $redeploy(Component\ c, Node\ n)$: this action moves the identified component c to the node n. Such a refactoring action is aimed at improving the utilization of the node where the component c was deployed.

- $split(Component\ c, Integer\ i, Node[]\ n)$: this action equally distributes (modulo i) the connections of the component c between the latter and several new components that are uniformly deployed on the set of nodes n. Such a refactoring action is aimed at reducing the number of connections of c in an efficient way.

- $mirror(Component\ c, Integer\ i, Node[]\ n)$: this action creates i copies (mirrors) of the c component. Mirrors are uniformly deployed on the set of nodes n. This means that the incoming workload is equally distributed between the component c and its mirrors.

- $replace(Component\ c, Float\ f, Service\ s)$: this action replaces the component c with a new component having a resource demand for the service s equal to $f * r(c, s)$, where $r(c, s)$ is the resource demand of c for s, and f is a scale factor in the interval $]0, 1[$.

The application of refactoring actions can be limited by pre-existing (functional or non-functional) requirements. Examples of functional requirements are legacy ones that disable components to be split and redeployed. Examples of non-functional requirements are security issues that do not allow the redeployment of software components due to the critical information they manage. For the sake of automation, such requirements should be pre-defined so that the whole process can take into account them and preventively excluding infeasible refactoring actions.

For the sake of simplicity, in the following detection/refactoring step, we separately focus on the Blob, CPS, and the TJ antipatterns.

Blob - Table 11 reports some thresholds involved in the Blob antipattern specification.

Table 11: Thresholds binding for the Blob antipattern.

Threshold	Value
$Th_{maxConnects}$	5
$Th_{maxHwUtil}$	90%

With these numerical values, one occurrence of Blob is detected, i.e., the *MoviesController* component. The *MoviesCatalog* component is not a Blob because it has a number of connections lower than $Th_{maxConnects}$ (i.e., 5). Furthermore, although the *BooksController* component has a number of connections larger than $Th_{maxConnects}$, the utilization of the node where it is deployed (i.e., *BooksControlNode*, whose utilization is 87.4%) is not larger than the $Th_{maxHwUtil}$ threshold (i.e., 90%). For similar reasons, *BooksCatalog* and *Database* components are not Blobs.

As refactoring action for the *MoviesController* Blob, we applied

$$redeploy(MoviesController,\ MoviesDispatcherNode)$$

This refactoring leads to a response time of 1.98 seconds, which still does not satisfies the requirement (1.83 seconds). Hence, the preliminary Blob detection is not effective with respect to the requirement of *MakePurchase* response time, maybe due to the fact that more significant antipatterns occur in the system.

CPS - Table 12 reports some thresholds involved in the CPS antipattern specification.

Table 12: Thresholds binding for the Concurrent Processing Systems antipattern.

Threshold	Value
Th_{maxQL}	8
$Th_{maxCpuUtil}$	90%
$Th_{minCpuUtil}$	50%

With these numerical values, two occurrences of CPS are detected, i.e., (*MoviesControlNode*, *BooksDispatcherNode*) and (*MoviesControlNode*, *WebServerNode*) pairs of hardware nodes.

In case of (*MoviesControlNode*, *BooksDispatcherNode*) occurrence, we applied

$$redeploy(getCriticalComponent(MoviesControlNode),\ BooksDispatcherNode)$$

where $getCriticalComponent(Node\ n)$ returns the most critical component, i.e., the one having the highest resource demand among all the components deployed on n, and in particular:

$$getCriticalComponent(Node\ n) = \begin{cases} MoviesController \\ \quad \text{if } n = MoviesControlNode \\ BooksController \\ \quad \text{if } n = BooksControlNode \end{cases}$$

This refactoring leads to a response time of 1.66 seconds, which satisfies the requirement (1.83 seconds).

Similarly, in case of (*MoviesControlNode*, *WebServerNode*) occurrence, we applied

$$redeploy(getCriticalComponent(MoviesControlNode), WebServerNode)$$

This refactoring leads to a response time of 1.67 seconds, which satisfies the requirement (1.83 seconds).

Hence, both the refactoring actions applied to remove the CPS antipattern are beneficial for fulfilling the requirement of 1.83 seconds defined on the *MakePurchase* response time.

TJ - Table 13 reports some thresholds involved in the TJ antipattern specification. With this numerical value one occurrence of TJ is detected, i.e., the *BrowseCatalog* service.

Table 13: Thresholds binding for the Traffic Jam antipattern.

Threshold	Value
$Th_{OpRtVar}$	0.25

Figure 7a illustrates an excerpt of the response time (observed during simulation) of the *BrowseCatalog* service, where we highlight the TJ antipattern occurrence. On the x-axis the simulation time is reported and on the y-axis the response time of the service is depicted. The trend of the average response time for the *BrowseCatalog* service is shown. We obtained this trend by dividing the simulation time in intervals of 50 seconds and, for each interval, we calculated the average response time of the observed completions. Hence, we drew the *average trend* by considering the calculated average response time as constant in the referring interval to obtain the piecewise linear function, i.e., the solid line of Figure 7a. The average trend is observed after 150 seconds because the initial simulation values may be misleading due to the setting of simulation seeds.

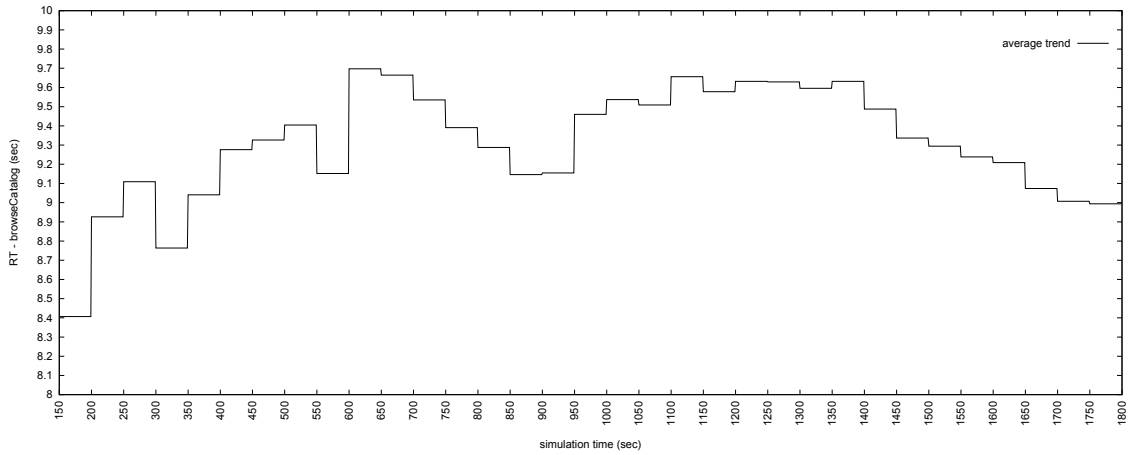
We observe several intervals with the occurrence of the TJ antipattern, i.e., $[150,200]$, $[300,350]$, $[600,650]$, $[900,950]$. For example, in the interval $[900,950]$ we observe that the *BrowseCatalog* service shows the following features: (a) it has a quite stable value of its response time along previous observation time slots up to 950 seconds of simulation time; (b) it has an increasing value of response time in the intervals $[900,950]$ and $[950,1000]$ (in fact a peak is shown: RT(*BrowseCatalog*, 900) = 9.14 seconds and RT(*BrowseCatalog*, 950) = 9.46 seconds by giving raise to a gap of 0.32 seconds that is larger than the $Th_{OpRtVar}$ threshold value set to 0.25 seconds); and (c) it has a quite stable value of its response time after 1000 seconds of simulation time (in fact RT(*BrowseCatalog*, 1000) = 9.53 seconds).

As refactoring action for the unique TJ occurrence, we applied

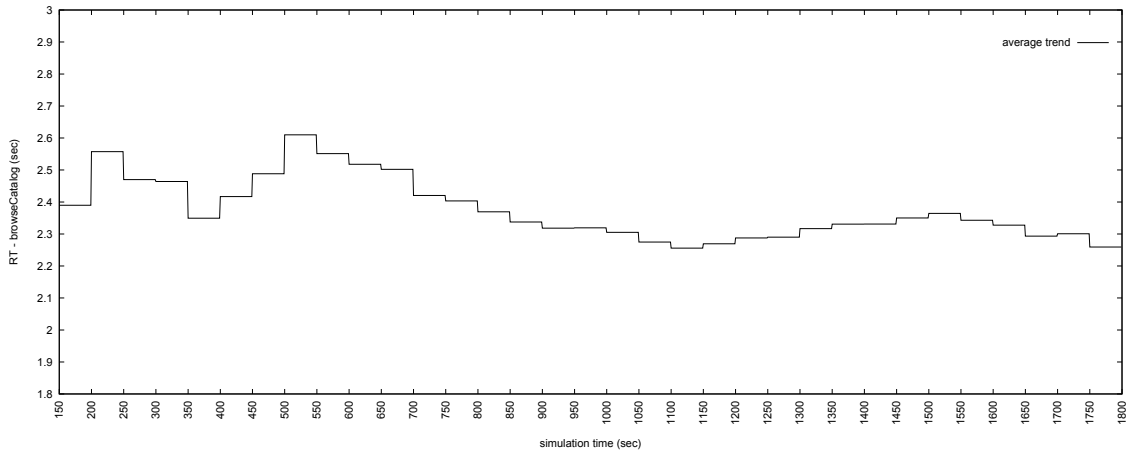
$$replace(getCriticalComponent(n), 0.25, BrowseCatalog)$$

for each node n in the deployment view. The replacement is enabled by different implementations of the same components that, however, may require other amounts of resource demands. The replacement of software components may incur different component costs, thus inducing a quality trade-off issue. For the sake of simplicity, we do not consider this issue here.

Figure 7b illustrates the performance improvement that we observe for the response time of the *BrowseCatalog* service, i.e., for the average trend in the different time slots. Similarly to Figure 7a, the average trend is observed after 150 seconds. We find that the maximum slope is achieved for the interval $[450, 500]$ of simulation time where there is a difference of 0.12 seconds in the response time of the *BrowseCatalog* service, i.e., lower than the $Th_{OpRtVar}$ threshold value (see Table 13). The average response time is 2.34 seconds, and the requirement (5 seconds) is satisfied.



(a) *Traffic Jam* antipattern occurrence.



(b) Performance improvement due to the *Traffic Jam* antipattern solution.

Figure 7: ECS- a deeper analysis for the response time of the *BrowseCatalog* service.

5 Detection and refactoring of antipatterns: sensitivity analysis vs. thresholds

In this section, we show the impact of thresholds on the capability of detecting and refactoring performance antipatterns. For this goal, we perform antipattern detection and refactoring on the ECS introduced in the previous section, while varying the numerical values of several thresholds. We quantify the threshold variations with the support of precision and recall metrics.

5.1 Precision and Recall for the performance

Precision and recall are well-known metrics aimed at quantifying the effectiveness of a technique for pattern recognition or information retrieval [FB92]. In simple terms, high recall means that the technique has returned most of the relevant results, while high precision means that it has returned substantially more relevant results than irrelevant ones.

In the area of pattern/antipattern detection, these metrics have been very useful to compare different techniques in quantitative terms [MGDM10]. When analyzing the results of pattern/antipattern detection, a pattern/antipattern occurrence can be classified into one of four categories: *true-positive* (TP: correctly found), *false-positive* (FP: incorrectly found), *true-negative* (TN: correctly unfound), and *false-negative* (FN: incorrectly unfound). Two common metrics of measuring the accuracy of detection results are then precision and recall. Precision is the ratio of correctly found to all found occurrences and equals to $TP/(TP+FP)$. Recall is the ratio of correctly found to all correct occurrences and equals to $TP/(TP+FN)$.

To apply these metrics in the context of performance antipatterns, we first must refine the concepts of true and false positives in such intrinsically stochastic context. In fact, even if a performance antipattern represents a bad design practice that may adversely affect the system performance, the removal of an antipattern does not certainly lead to improve the system performance because refactoring actions, applied to remove it, might introduce performance problems somewhere else in the system, and these problems emerge only after solving the refactored performance model.

In this paper, we associate the recall metrics to the detection activity and the precision metrics to the refactoring activity, as follows. The recall is defined as the ratio between the number of detected performance antipatterns and the number of existing performance antipatterns. We define the latter quantity as the number of all performance antipatterns that can be detected while varying the antipattern thresholds within predefined ranges.

As mentioned above, to distinguish between true and false positives, we should first observe the effect of an antipattern removal on the system performance. Therefore, the precision is defined as the ratio between the number of detected performance antipatterns that actually improve the system performance once removed and the number of detected performance antipatterns.

When analyzing the results of performance antipatterns detection and refactoring, a performance antipattern occurrence can be classified into one of four categories: *true-positive* (TP: correctly found, i.e., the detected antipatterns whose refactoring is beneficial), *false-positive* (FP: incorrectly found, i.e., the detected antipatterns whose refactoring is not beneficial), *true-negative* (TN: correctly unfound, i.e., the undetected antipatterns whose refactoring is not beneficial) and *false-negative* (FN: incorrectly unfound, i.e., the undetected antipatterns whose

refactoring is beneficial). Because the detection process provides different results depending on thresholds binding, we proceed with the following definitions. Recall is the ratio between all the occurrences found with thresholds binding and all found occurrences while varying the antipattern thresholds, thus it equals to $(TP+FP)/(TP+FP+TN+FN)$. Hence, a recall score of 4/5 means that 4 occurrences were detected over 5 occurrences that are found while varying the antipattern thresholds within predefined ranges. Precision is the ratio of correctly found to all found occurrences and equals $TP/(TP+FP)$. Hence, a precision score of 2/4 means that 2 found occurrences were beneficial for the system performance.

In the next section we show the application of these metrics to the ECS example.

5.2 Precision and Recall Applied to ECS

In this section, we present the experimentation that we conducted while varying thresholds of Blob and CPS antipatterns, first separately and then in an aggregate way. We report the influence of these variations on precision and recall metrics. We do not consider the TJ antipattern because our ECS does not present multiple TJ occurrences, thus precision and recall would be trivial for this case.

B.I Impact of Blob Thresholds

Regarding the Blob antipattern, we varied $Th_{maxConnects}$, initially set to 5, in the interval $[4, 6]$ and $Th_{maxHwUtil}$, initially set to 90%, in the interval $[85\%, 95\%]$. Such variations lead 4 Blob occurrences to emerge in the ECS, that are *MoviesController*, *MoviesCatalog*, *BooksController*, and *BooksCatalog*.

Table 14 summarizes the set of detected Blob occurrences while varying the considered thresholds and reports the recall for each variation. The first column (i.e., #) identifies the variation. The column *From* shows the initial value for each considered threshold whereas the column *To* shows the value that the threshold assumes after the variation has been applied. The "−" symbol in the *To* column indicates that no variation has been made for the corresponding threshold value.

Table 14: Blob thresholds variations vs. detection vs. recall.

#	Variation				Detected Blobs	Recall
	$Th_{maxConnects}$		$Th_{maxHwUtil}$			
	From	To	From	To		
1	5	6	90%	-	{}	0/4
2	5	-	90%	95%	{}	0/4
3	5	4	90%	-	{*MoviesController*, *MoviesCatalog*}	2/4
4	5	-	90%	85%	{*MoviesController*, *BooksController*}	2/4
5	5	4	90%	85%	{*MoviesController*, *MoviesCatalog*, *BooksController*, *BooksCatalog*}	4/4

By increasing $Th_{maxConnects}$ from 5 to 6 and–or $Th_{maxHwUtil}$ from 90% to 95% no Blobs are detected and the recall is obviously 0. These thresholds represent upper bounds and it is useless to explore further variations in this direction. On the contrary, while decreasing thresholds, we observe that the number of detected antipatterns (thus the recall too) increases. In fact, by

decreasing one of them (i.e., $Th_{maxConnects}$ from 5 to 4 or $Th_{maxHwUtil}$ from 90% to 85%) a new Blob occurrence is detected in addition to the *MoviesController* component, resulting in a recall of 0.5. By decreasing both $Th_{maxConnects}$ from 5 to 4 and $Th_{maxHwUtil}$ from 90% to 85% (variation #5), three new Blob occurrences in the system are detected, resulting in a recall of 1.

For each Blob occurrence *blob*, we applied the *redeploy* and *split* refactoring actions as it follows:

– $redeploy(blob,\ hwNode)$;
– $split(blob,\ ceiling(getNumConnects(blob)/Th_{maxConnects}),\ \{hwNode\})$;

where $ceiling(Float\ f)$ returns the smallest integer not less than f, $getNumConnects(Component\ c)$ returns the number of connections of the component c, and

$$hwNode = \begin{cases} MoviesDispatcherNode \\ \qquad \text{if } blob = \{MoviesController, MoviesCatalog\} \\ BooksDispatcherNode \\ \qquad \text{if } blob = \{BooksController, BooksCatalog\} \end{cases}$$

Table 15 summarizes the response times for the *MakePurchase* service while varying Blob thresholds in the most significative ways (i.e., #3, #4, and #5) and applying the *redeploy* and *split* refactoring actions. The table also reports corresponding precisions. In the remainder of the paper, shaded entries in tables represent beneficial refactorings, i.e., the ones that result in a response time lower or equal than 1.83 seconds.

Table 15: Blob thresholds variations vs. refactoring vs. precision.

#	Average response time after Blob refactorings								Precision
	MoviesController		*MoviesCatalog*		*BooksController*		*BooksCatalog*		
3	*redeploy* 1.98	*split* 1.83	*redeploy* 4.47	*split* 1.66	-		-		2/2
4	*redeploy* 1.98	*split* 1.83	-		*redeploy* 1.84	*split* 1.91	-		1/2
5	*redeploy* 1.98	*split* 1.83	*redeploy* 4.47	*split* 1.66	*redeploy* 1.84	*split* 1.91	*redeploy* 1.86	*split* 1.85	2/4

Let us now focus on each row of Table 15.

#3: After redeploying *MoviesCatalog*, an average response time of 4.47 seconds is obtained. This is the worst case because the resource demand of *MoviesCatalog* is too heavy for *DispatcherMoviesNode* that already hosts *MoviesDispatcher* and *UserController*. This results in a saturation of *DispatcherMoviesNode* in the refactored model and the requirement on the *MakePurchase* service is not fulfilled.

By splitting *MoviesCatalog*, an average response time of 1.66 seconds is obtained. This response time represents an improvement compared to the one deriving from splitting *MoviesController* and is due to the different resource demand of the involved components. Moreover, we can guarantee that the components involved in the splitting action have a number of connections lower than the modified $Th_{maxConnects}$ threshold (i.e., 4). In fact, in the refactored model, they

only have 2 connections and this falsifies the clause related to the number of connections in the Blob antipattern definition. Hence, the Blob occurrence identified by *MoviesCatalog* is removed and the requirement on the *MakePurchase* service (1.83 seconds) is fulfilled.

The precision of variation #3 is equal to 1 because, for each detected antipattern, it exists at least a refactoring action that removes the antipattern and satisfies the requirement, i.e., the *split* refactoring action.

#4: After redeploying *BooksController*, an average response time of 1.84 seconds is obtained. This response time represents an improvement with respect to the one deriving from the first redeployment action but it is not sufficient for satisfying the requirement. Anyhow, we cannot guarantee that all the nodes involved in the redeployment action have an utilization lower than the modified $Th_{maxHwUtil}$ threshold (i.e., 85%). Hence, we need a further performance analysis step on the refactored model to verify if the corresponding clause in the Blob antipattern definition has been falsified.

By splitting *BooksController*, an average response time of 1.91 seconds is obtained. As in the case of *MoviesCatalog* splitting, we can guarantee that each component involved in the splitting action has a number of connections lower than the original $Th_{maxConnects}$ threshold (i.e., 5). In fact, in the refactored model, they have at most 3 connections and this number falsifies the clause related to the number of connections in the Blob antipattern definition. Hence, the Blob occurrence identified by *MoviesCatalog* is removed but the requirement on the *MakePurchase* service (1.83 seconds) is not fulfilled.

The precision of variation #4 is equal to 0.5, because only for 1 of 2 detected antipatterns there exists at least a refactoring action that removes the antipattern and satisfies the requirement, i.e., the *split* refactoring action applied to the antipattern identified by *MoviesController*.

#5: After redeploying *BooksCatalog*, an average response time of 1.86 seconds is obtained and we need a further performance analysis step on the refactored model to verify if the clause concerning $Th_{maxHwUtil}$ in the Blob antipattern definition has been falsified. By splitting *BooksCatalog*, the average response time is 1.85 seconds and we can guarantee the Blob removal by falsifying the clause related to $Th_{maxNumConnects}$ in the Blob antipattern definition. However, both these refactoring actions are not sufficient to satisfy the requirement on the response time of *MakePurchase* (1.83 seconds).

The precision of variation #5 is equal to 0.5, because only for 2 of 4 detected antipatterns there exists at least a refactoring action that removes the antipattern and satisfies the requirement, i.e., the *split* refactoring action applied to the antipatterns identified by *MoviesController* and *MoviesCatalog*.

It is interesting to note that the best precision is achieved by variation #3, that is the one having the lowest recall (together with variation #4) among the variations. Variation #3 detects exactly the two Blobs whose removal leads to a response time that satisfies the requirement for the *MakePurchase* service (1.83 seconds). As opposite, the best recall is achieved by variation #5, i.e., the one having the lowest precision (together with variation #4). Variation #4 detects all the four Blob occurrences in the ECS; because only two occurrences can be removed satisfying the requirement for the *MakePurchase* service, variation #4 has the lowest precision.

B.II Impact of Concurrent Processing Systems Thresholds

Regarding the CPS antipattern, we varied Th_{maxQL}, initially set to 8, in the interval [5, 9], $Th_{maxCpuUtil}$ in the interval [85%, 95%] starting from 90%, and $Th_{minCpuUtil}$ in the interval [5%, 55%] starting from 50%. Such variations lead 5 CPS occurrences to emerge in the ECS[4], that are (*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*), (*BCN*, *BDN*), and (*BCN*, *WSN*).

Table 16 summarizes the set of detected CPS occurrences while varying the considered thresholds and reports the recall for each variation.

Table 16: CPS thresholds variations vs. detection vs. recall.

#	Variation Th_{maxQL} From	Th_{maxQL} To	$Th_{minCpuUtil}$ From	$Th_{minCpuUtil}$ To	$Th_{maxCpuUtil}$ From	$Th_{maxCpuUtil}$ To	Detected CPS	Recall
1	8	9	50%	-	90%	-	{}	0/5
2	8	-	50%	-	90%	95%	{}	0/5
3	8	-	50%	5%	90%	-	{}	0/5
4	8	-	50%	55%	90%	-	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*)}	3/5
5	8	5	50%	-	90%	85%	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*BCN*, *BDN*), (*BCN*, *WSN*)}	4/5
6	8	5	50%	55%	90%	85%	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*), (*BCN*, *BDN*), (*BCN*, *WSN*)}	5/5

By increasing Th_{maxQL} from 8 to 9 and–or $Th_{maxCpuUtil}$ from 90% to 95%, no CPS are detected and the recall is obviously 0. These thresholds represent upper bounds and it is useless to explore further variations in this direction. Similarly, by decreasing $Th_{minCpuUtil}$ from 50% to 5%, no CPS are detected, thus the recall is 0. This latter threshold represents a lower bound, hence it is again useless to explore further variations.

On the contrary, while decreasing upper bound thresholds and–or increasing the lower bound threshold, we observe that the number of detected antipatterns (thus the recall too) increases. In fact, by increasing $Th_{minCpuUtil}$ from 50% to 55% (i.e., variation #4), a new CPS occurrence, i.e., (*MCN*, *MDN*), is detected in addition to the ones detected during the first antipatterns detection step, resulting in a recall of 3/5.

Furthermore, by decreasing one or both of them (i.e., Th_{maxQL} from 8 to 5 and–or $Th_{maxCpuUtil}$ from 90% to 85%), new CPS occurrences are detected. In particular, we observe that: (i) by decreasing Th_{maxQL} from 8 to 5 and $Th_{maxCpuUtil}$ from 90% to 85% (i.e., variation #5), two new CPS occurrences, i.e., (*BCN*, *BDN*) and (*BCN*, *WSN*), are detected in addition to the ones detected during the first detection step and the recall is 4/5; (ii) by decreasing $Th_{maxCpuUtil}$ from

[4] For the sake of the readability, we use acronyms to name nodes involved in CPS occurrences: *MCN* for *MoviesControlNode*, *MDN* for *MoviesDispatcherNode*, *BCN* for *BooksControlNode*, *BDN* for *BooksDispatcherNode*, and *WSN* for *WebServerNode*.

90% to 85% and by increasing $Th_{minCpuUtil}$ from 50% to 55% (i.e., variation #6) at the same time, all the five CPS occurrences are detected, resulting in a recall of 1.

For each CPS occurrence $(hwNode_1, hwNode_2)$, we applied the *mirror* and *redeploy* refactoring actions as follows:

– $mirror(getCriticalComponent(hwNode_1),\ 2,\ \{hwNode_2\})$;
– $redeploy(getCriticalComponent(hwNode_1),\ hwNode_2)$;

where $getCriticalComponent(Node\ n)$ is the previously defined function.

Table 17 summarizes the response times for the *MakePurchase* service while varying CPS thresholds in the most significative ways (i.e., #4, #5, and #6) and applying the *redeploy* and *mirror* refactoring actions as described. The table also reports corresponding precisions.

Table 17: CPS thresholds variations vs. refactoring vs. precision.

#	Average response time after CPS refactorings										Precision
	(*MCN*, *BDN*)		(*MCN*, *WSN*)		(*MCN*, *MDN*)		(*BCN*, *BDN*)		(*BCN*, *WSN*)		
4	*mirror* 1.66	*redeploy* 1.61	*mirror* 1.67	*redeploy* 1.64	*mirror* 1.82	*redeploy* 2.96	-		-		3/3
5	*mirror* 1.66	*redeploy* 1.61	*mirror* 1.67	*redeploy* 1.64	-		*mirror* 1.86	*redeploy* 1.84	*mirror* 1.87	*redeploy* 1.85	2/4
6	*mirror* 1.66	*redeploy* 1.61	*mirror* 1.67	*redeploy* 1.64	*mirror* 1.82	*redeploy* 2.96	*mirror* 1.86	*redeploy* 1.84	*mirror* 1.87	*redeploy* 1.85	3/5

Both the *mirror* and the *redeploy* refactoring actions lead to the fulfillment of the requirement on the *MakePurchase* service, if applied to the (*MCN*, *BDN*) or (*MCN*, *WSN*) CPS occurrences. On the contrary, neither of the two applicable refactoring actions fullfill the requirement, if applied to (*BCN*, *BDN*) or (*BCN*, *WSN*). Regarding the (*MCN*, *MDN*) occurrence, only the *mirror* refactoring action is beneficial for the requirement, whereas the *redeploy* one leads to an average response time of 2.96 seconds, due to the resource demand of the *MoviesController* component that cannot be handled by *MoviesDispatcherNode*.

As previously stated, we need a further performance analysis step on the refactored model to verify if the clauses related to $Th_{minCpuUtil}$ and $Th_{maxCpuUtil}$ in the CPS antipattern definition have been falsified after each refactoring.

In Table 17, we observe that the precision of variation #4 is equal to 1, because for each detected CPS occurrence there exists at least a refactoring action that removes the antipattern and satisfies the requirement for the *MakePurchase* service. Variation #5 has a precision equal to 0.5, because 2 of the 4 detected CPS occurrences can be removed, satisfying the requirement. Finally, although variation #6 detects all the 5 CPS occurrences in the ECS, only 3 of them are solvable in a beneficial way with respect to the requirement.

It is worth to notice that, similarly to the Blob occurrences, the best precision is achieved by the variation with the lowest recall, i.e., variation #4. Variation #4 detects exactly the three CPS whose removal leads to a response time that satisfies the requirement for the *MakePurchase* service (1.83 seconds). As opposite, although variation #6 detects the latter CPS occurrences, it also detects the remaining two ones whose removal does not satisfies the requirement, reducing precision.

B.III Aggregating Thresholds

In this section, we aggregate thresholds for the Blob and CPS antipatterns, because both the $Th_{maxHwUtil}$ of the Blob and the $Th_{maxCpuUtil}$ of the CPS represent upper bounds for the utilization of hardware devices and they vary in the same interval of values (i.e., [85%, 95%]). Hence, we can unify $Th_{maxCpuUtil}$ to $Th_{maxHwUtil}$ and vary the new set of aggregated thresholds as we varied thresholds by considering one antipattern at a time.

We remind that the ECS contains 4 Blob occurrences, i.e., *MoviesController*, *MoviesCatalog*, *BooksController*, and *BooksCatalog*, and 5 CPS occurrences, i.e., (*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*), (*BCN*, *BDN*), and (*BCN*, *WSN*).

Table 18 summarizes the set of detected Blob and CPS occurrences while varying the new set of thresholds and reports the recall for each variation.

Table 18: Blob and CPS thresholds variations vs. aggregated detection vs. aggregated recall.

#	Variation								Detected Blobs	Detected CPS	Recall
	$Th_{maxNumConnects}$		Th_{maxQL}		$Th_{minCpuUtil}$		$Th_{maxHwUtil}$				
	From	To	From	To	From	To	From	To			
1	5	6	8	-	50%	-	90%	-	{}	{}	0/9
2	5	-	8	9	50%	-	90%	-	{}	{}	0/9
3	5	-	8	-	50%	5%	90%	-	{}	{}	0/9
4	5	-	8	-	50%	-	90%	95%	{}	{}	0/9
5	5	4	8	-	50%	-	90%	-	{*MoviesController*, *MoviesCatalog*}	{}	2/9
6	5	-	8	-	50%	-	90%	85%	{*MoviesController*, *BooksController*}	{}	2/9
7	5	-	8	-	50%	55%	90%	-	{}	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*)}	3/9
8	5	4	8	-	50%	-	90%	85%	{*MoviesController*, *MoviesCatalog*, *BooksController*, *BooksCatalog*}	{}	4/9
9	5	4	8	-	50%	55%	90%	-	{*MoviesController*, *MoviesCatalog*}	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*)}	5/9
10	5	-	8	5	50%	-	90%	85%	{*MoviesController*, *BooksController*}	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*BCN*, *BDN*), (*BCN*, *WSN*)}	6/9
11	5	-	8	5	50%	55%	90%	85%	{*MoviesController*, *BooksController*}	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*), (*BCN*, *BDN*), (*BCN*, *WSN*)}	7/9
12	5	4	8	5	50%	-	90%	85%	{*MoviesController*, *MoviesCatalog*, *BooksController*, *BooksCatalog*}	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*BCN*, *BDN*), (*BCN*, *WSN*)}	8/9
13	5	4	8	5	50%	55%	90%	85%	{*MoviesController*, *MoviesCatalog*, *BooksController*, *BooksCatalog*}	{(*MCN*, *BDN*), (*MCN*, *WSN*), (*MCN*, *MDN*), (*BCN*, *BDN*), (*BCN*, *WSN*)}	9/9

Table 19 summarizes response times for the *MakePurchase* service while varying Blob-CPS thresholds in the most significative ways (i.e., from #5 to #13) and applying the *mirror*, *redeploy*, and *split* refactoring actions as previously described[5]. For each antipattern occurrence, the table only reports the most beneficial refactoring actions between the three applicable ones. The table also reports corresponding precisions.

Table 19: Blob and CPS thresholds variations vs. refactoring vs. precision.

#	Average response time after Blob-CPS refactorings									Precision
	Blobs				CPS					
	MCo	*MCa*	*BCo*	*BCa*	(*MCN*, *BDN*)	(*MCN*, *WSN*)	(*MCN*, *MDN*)	(*BCN*, *BDN*)	(*BCN*, *WSN*)	
5	*split* 1.83	*split* 1.66	-	-	-	-	-	-	-	2/2
6	*split* 1.83	-	*redeploy* 1.84	-	-	-	-	-	-	1/2
7	-	-	-	-	*redeploy* 1.61	*redeploy* 1.64	*mirror* 1.82	-	-	3/3
8	*split* 1.83	*split* 1.66	*redeploy* 1.84	*split* 1.85	-	-	-	-	-	2/4
9	*split* 1.83	*split* 1.66	-	-	*redeploy* 1.61	*redeploy* 1.64	*mirror* 1.82	-	-	5/5
10	*split* 1.83	-	*redeploy* 1.84	-	*redeploy* 1.61	*redeploy* 1.64	-	*redeploy* 1.84	*redeploy* 1.85	3/6
11	*split* 1.83	-	*redeploy* 1.84	-	*redeploy* 1.61	*redeploy* 1.64	*mirror* 1.82	*redeploy* 1.84	*redeploy* 1.85	4/7
12	*split* 1.83	*split* 1.66	*redeploy* 1.84	*split* 1.85	*redeploy* 1.61	*redeploy* 1.64	*redeploy* 1.84	*redeploy* 1.85	-	4/8
13	*split* 1.83	*split* 1.66	*redeploy* 1.84	*split* 1.85	*redeploy* 1.61	*redeploy* 1.64	*mirror* 1.82	*redeploy* 1.84	*redeploy* 1.85	5/9

The precision of variations #5, #7, and #9, is equal to 1, because for each detected CPS occurrence there exists at least a refactoring action that removes the antipattern occurrence and satisfies the requirement. The variation with the highest precision among the ones that detect at least one occurrence of both the Blob an the CPS antipatterns, i.e., variation #9, is also the one with the lowest recall, i.e., 5/9, because that variation detects exactly the only two Blobs and the only three CPS whose removal leads to a response time that satisfies the requirement for the *MakePurchase* service (1.83 seconds). Variations from #10 to #13 have a precision in the interval [0.5, 4/7], because for all detected CPS occurrences there does not exists a refactoring action that removes the antipattern occurrence and satisfies the requirement for the *MakePurchase* service.

6 Discussion

Several observations regarding the antipattern detection and refactoring derive from our experimentation.

Thresholds vs. antipatterns removal - If a refactoring action refers to a threshold related to a design feature (e.g., number of connections), we can ensure that its application leads to

[5] For the sake of table readability, we use acronyms to name components involved in Blob occurrences: *MCo* for *MoviesController*, *MCa* for *MoviesCatalog*, *BCo* for *BooksController*, and *BCa* for *BooksCatalog*.

the removal of the antipattern occurrence. On the other hand, if a refactoring action refers to a threshold related to a performance index (e.g., hardware nodes utilization or throughput, service response time, etc.), we cannot ensure that its application leads to the actual removal of the antipattern occurrence. A further performance analysis step for the refactored model is needed. For example, in Table 15 (see variation #5), we have shown that the refactoring action referring to a threshold related to a performance index (i.e., *redeploy*) worsen (4.47 seconds) the *MakePurchase* response time, hence the antipattern is not actually removed. On the contrary, the refactoring action referring to a threshold related to a design feature (i.e., *split*) improves (1.66 seconds) the *MakePurchase* response time and the antipattern is actually removed.

Because the specification of some antipatterns only contains thresholds related to performance indices, we have experienced that it is more difficult to refactor such antipatterns rather than the ones referring also to design features.

Furthermore, the value of thresholds related to design features (e.g., number of connections) influences the refactoring actions to put in place to remove an antipattern. For example, $Th_{maxConnects}$ refers to the maximum number of connections for software components, thus its value induces the *split* refactoring action to generate a specific number of new components that can be deployed according to different deployment strategies.

Recall vs. precision - In our experimentation, we found that the highest precision is achieved with thresholds values leading to the lowest recall (i.e., variation #3 in Tables 14 and 15, variation #4 in Tables 16 and 17, and variation #9 in Tables 18 and 19). This precision is due to the set of detected antipatterns that may contain occurrences whose removal is not helpful for requirement(s) fulfillment. For example, in Tables 16 and 17, we observe that: (i) variation #9 detects five of nine occurrences (recall = 5/9) that satisfy the requirement (precision = 1); (ii) variation #13 detects all the nine occurrences (higher recall, equal to 1), but the new ones are not beneficial (lower precision, equal to 5/9).

Restrictions in refactoring applicability - The application of refactorings can be restricted by functional or non-functional requirements. Example of functional requirements may be legacy components that might restrict the set of applicable refactoring actions. In our example, let us suppose that *MoviesController* cannot be refactored because it is a legacy component, then the *split* action on that component cannot be applied anymore and we can only apply the *redeploy* action, that does not result in a requirement fulfillment. Example of non-functional requirements may be budget limitations that do not allow to adopt an antipattern solution due to its extremely high cost. Many other examples can be provided of requirements that (implicitly or explicitly) may affect the antipattern solution activity.

7 Conclusion

In this paper, we have analyzed the influence of numerical thresholds on the capability and effectiveness of detecting and refactoring performance antipatterns. In particular, we have defined the recall and precision of detection rules and refactoring actions respectively and we have applied them to a case study in the e-commerce domain.

It is certainly of great interest to extend the experiment reported here to other performance antipatterns. More complex examples shall be considered and domain-specific and system-specific

characteristics could be also exploited for this goal, because they may have an impact on threshold values.

We also intend to introduce confidence values that may be associated to antipattern occurrences to quantify the probability that numerical threshold values support the actual antipattern presence. Furthermore, some fuzziness can be introduced for the evaluation of the threshold values [Mar04, OKAG10, SCK02] thus to make antipattern detection rules more flexible.

Acknowledgements: This work has been partially supported by the European Office of Aerospace Research and Development (EOARD), Grant/Cooperative Agreement (Award no. FA8655-11-1-3055), and by the VISION European Research Council Starting Grant (ERC-240555).

Bibliography

[AC13] D. Arcelli, V. Cortellessa. Software model refactoring based on performance analysis: better working on software or performance side? In Buhnova et al. (eds.), *FESCA*. EPTCS 108, pp. 33–47. 2013.

[ACT12] D. Arcelli, V. Cortellessa, C. Trubiani. Antipattern-based model refactoring for software performance improvement. In *ACM SIGSOFT International Conference on Quality of Software Architectures (QoSA)*. Pp. 33–42. 2012.

[BDC02] M. Bernardo, L. Donatiello, P. Ciancarini. Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. In *Performance Evaluation of Complex Systems: Techniques and Tools, Tutorial Lectures, Performance*. Pp. 236–260. 2002.

[BDIS04] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. Software Eng.* 30(5):295–310, 2004.

[BKR09] S. Becker, H. Koziolek, R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82(1):3–22, 2009.

[CDDT12] V. Cortellessa, M. De Sanctis, A. Di Marco, C. Trubiani. Enabling Performance Antipatterns to arise from an ADL-based Software Architecture. In *Joint Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA*. 2012.

[CDE+10] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, C. Trubiani. Digging into UML models to remove performance antipatterns. In *ICSE Workshop Quovadis*. Pp. 9–16. 2010.

[CDT12] 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*, 2012. DOI: 10.1007/s10270-012-0246-z.

[CM02] V. Cortellessa, R. Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Sci. Comput. Program.* 44(1):101–129, 2002.

[CMI11] V. Cortellessa, A. D. Marco, P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[CS11] G. Casale, G. Serazzi. Quantitative system evaluation with Java modeling tools. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*. ICPE '11, pp. 449–454. ACM, New York, NY, USA, 2011. doi:10.1145/1958746.1958813 http://doi.acm.org/10.1145/1958746.1958813

[FB92] W. B. Frakes, R. Baeza-Yates (eds.). *Information retrieval: data structures and algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[HT07] M. Harman, L. Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. GECCO '07, pp. 1106–1113. ACM, New York, NY, USA, 2007. doi:10.1145/1276958.1277176 http://doi.acm.org/10.1145/1276958.1277176

[Jai91] R. Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. *SIGMETRICS Performance Evaluation Review* 19(2):5–11, 1991. doi:http://doi.acm.org/10.1145/122564.1045495

[KPGA12] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17(3):243–275, 2012.

[KVGS11] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. A. Sahraoui. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84(4):559–572, 2011.

[LKGS84] E. Lazowska, J. Kahorjan, G. S. Graham, K. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[Mar04] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *International Conference on Software Maintenance (ICSM)*. Pp. 350–359. 2004.

[MGDM10] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. L. Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Software Eng.* 36(1):20–36, 2010.

[MKBR10] A. Martens, H. Koziolek, S. Becker, R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *WOSP/SIPEW International Conference on Performance Engineering*. Pp. 105–116. 2010.

[MPN+12] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, J.-M. Jézéquel. Specification and Detection of SOA Antipatterns. In *International Conference on Service-Oriented Computing (ICSOC)*. Pp. 1–16. 2012.

[Obj05] Object Management Group (OMG). UML 2.0 Superstructure Specification. 2005. OMG Document formal/05-07-04.

[Obj09] Object Management Group (OMG). UML Profile for MARTE. 2009. OMG Document formal/08-06-09.

[OC08] M. O'Keeffe, M. í Cinnéide. Search-based refactoring for software maintenance. *J. Syst. Softw.* 81(4):502–516, Apr. 2008.
doi:10.1016/j.jss.2007.06.003
http://dx.doi.org/10.1016/j.jss.2007.06.003

[OKAG10] R. Oliveto, F. Khomh, G. Antoniol, Y.-G. Guéhéneuc. Numerical Signatures of Antipatterns: An Approach Based on B-Splines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. Pp. 248–251. 2010.

[PZ12] R. Peters, A. Zaidman. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *European Conference on Software Maintenance and Reengineering (CSMR)*. Pp. 411–416. 2012.

[RRPK12] D. Romano, P. Raila, M. Pinzger, F. Khomh. Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes. In *Working Conference on Reverse Engineering (WCRE)*. Pp. 437–446. 2012.

[SCK02] S. S. So, S. D. Cha, Y. R. Kwon. Empirical evaluation of a fuzzy logic-based software quality prediction model. *Fuzzy Sets Syst.* 127(2):199–208, Apr. 2002.
doi:10.1016/S0165-0114(01)00128-2
http://dx.doi.org/10.1016/S0165-0114(01)00128-2

[Smi07] C. U. Smith. Introduction to Software Performance Engineering: Origins and Outstanding Problems. In Bernardo and Hillston (eds.), *SFM*. Lecture Notes in Computer Science 4486, pp. 395–428. Springer, 2007.

[SSB06] O. Seng, J. Stammel, D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. GECCO '06, pp. 1909–1916. ACM, New York, NY, USA, 2006.
doi:10.1145/1143997.1144315
http://doi.acm.org/10.1145/1143997.1144315

[SW03] C. U. Smith, L. G. Williams. More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *International Computer Measurement Group Conference*. Pp. 717–725. 2003.

[TK11] C. Trubiani, A. Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In *International Conference on Performance Engineering (ICPE)*. Pp. 19–30. 2011.

[TSFB99] G. Travassos, F. Shull, M. Fredericks, V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Pp. 47–56. 1999.

[WFP07] C. M. Woodside, G. Franks, D. C. Petriu. The Future of Software Performance Engineering. In Briand and Wolf (eds.), *FOSE*. Pp. 171–187. 2007.

[Xu12] J. Xu. Rule-based automatic software performance diagnosis and improvement. *Perform. Eval.* 69(11):525–550, 2012.

[YM12] A. F. Yamashita, L. Moonen. Do code smells reflect important maintainability aspects? In *International Conference on Software Maintenance (ICSM)*. Pp. 306–315. 2012.