Influence of numerical thresholds on model-based detection and refactoring of performance antipatterns

Davide Arcelli, Vittorio Cortellessa, Catia Trubiani
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica (DISIM)
Università degli Studi dell'Aquila
L'Aquila, Italy
{davide.arcelli, vittorio.cortellessa, catia.trubiani}@univaq.it

***Abstract*—Performance antipatterns are well-known bad design practices that lead to software products suffering by poor performance. A certain 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.**

I. Introduction

In the software development domain there is a high interest in the early validation of performance requirements because it avoids late and expensive fix to consolidated software artifacts. Model-based approaches, pioneered under the name of Software Performance Engineering (SPE) by Smith [1], 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 [2].

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 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 [4]; (2) the *Model Solution* step takes as input a performance model and produces a set of performance indices [5]; (3) the *Performance Analysis Results Interpretation and Feedback Generation* macro step takes as input both the software model and the performance indices in order to detect possible performance problems[2], and it provides a refactored (annotated) software model where problems have been removed.

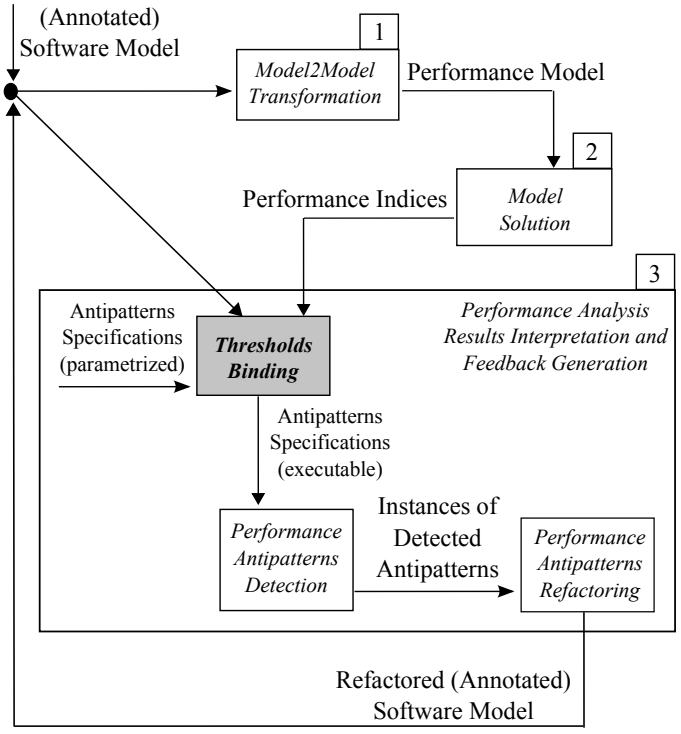


Fig. 1. Model-based software performance analysis process.

Few approaches have been recently introduced for this macro step [6], [7] among which we have been working on the detection and the refactoring of performance antipatterns [8], [9], [10].

Performance antipatterns [11] are well-known bad design practices that lead to software products suffering by poor performance, and they include solutions that let the software architects devise refactoring actions.

The macro step of Figure 1 has been detailed with the two main steps that we have envisaged in our approach, that are:

[1]Software model annotations support the performance analysis by specifying parameters like workload, resource demands, etc. [3]

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

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.

In order 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 instances 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 II provides some background on the thresholds introduced for the specification of performance antipatterns. Section III analyzes the thresholds influence through an example where experimental results arise the emerging issues in this context. Section IV concludes the paper by discussing the most challenging research topics in this area.

II. Thresholds in performance antipattern specification/representation

In literature several approaches have been introduced to specify and detect code smells and antipatterns [12], [13], [14], [15], [16]. They range from manual approaches, based on inspection techniques [17], to metric-based heuristics [18], [19], using rules and thresholds on various metrics [20] or Bayesian belief networks [21]. On the contrary, our approach intends to work at the design level and it can be applied early in the software life-cycle.

Since the performance antipatterns had been originally defined in natural language [11], 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 [22]. 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 [23].

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, and they additionally include thresholds on design features and performance indices. In fact, the characterization of antipattern parameters is related to design characteristics (e.g. *many* usage dependencies, *excessive* message traffic) and/or to performance results (e.g. *high*, *low* utilization) we interpret as thresholds.

Since we cannot avoid thresholds in antipatterns definition, the detection and the refactoring activities are heavily affected by the multiplicity and the estimation accuracy of thresholds an antipattern requires. In this direction, we are investigating the number and the type of thresholds the antipatterns' formalization requires.

Table I contains a list of performance antipatterns [11]. 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 [22]: 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 along the time, named *Multiple-values* performance antipatterns. Due to these characteristics, the performance indices needed to detect the latter type of antipatterns must be obtained via simulation or monitoring.

TABLE I. Overview of antipatterns thresholds.

Antipattern		Thresholds	
Type	Name	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 I we can notice 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) finally there is one antipattern (i.e. the More is Less) without thresholds because it lays on configuration parameters (database connections, 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 the whole approach, since they must be suitably estimated.

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 [22] we provided some heuristics to calculate these thresholds.

In the following we present the Blob performance antipattern example [11], i.e. the shaded entry of Table I. A Blob occurs when a component requires a *lot* of information from other ones, it generates *excessive* message traffic that

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

lead to *over utilize* the device on which it is deployed or the network involved in the communication. Table II reports the thresholds involved in the Blob specification [22]: two thresholds ($Th_{maxConnects}$, $Th_{maxMsgs}$) refer to design features, whereas the other ones ($Th_{maxHwUtil}$, $Th_{maxNetUtil}$) are related to performance indices.

TABLE II. THRESHOLDS SPECIFICATION FOR THE BLOB ANTIPATTERN.

	Threshold	Description
Design	$Th_{maxConnects}$	Maximum bound for the number of connections 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

Heuristics for Blob thresholds can be defined as follows [22]. $Th_{maxConnects}$ 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.

Note that the binding of some thresholds is intrinsically more difficult than others. For example, both The Ramp and the Traffic Jam antipatterns refer to thresholds representing the maximum feasible slope of the response time (or the throughput) observed in consecutive time slots, and these 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 the Traffic Jam antipatterns, such heuristics may 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.

III. INFLUENCE OF THRESHOLDS ON DETECTION AND REFACTORING

In this section we discuss the influence of thresholds on the detection and refactoring of performance antipatterns by means of a case study in the e-commerce domain. We first describe the E-Commerce System (ECS) software model and the numerical results obtained from its performance analysis, then emerging issues due to the variation of thresholds numerical values are presented.

A. An illustrative example

ECS is a web-based system that manages business data related to books and movies. We assume to have a multi-view model, composed by *Static*, *Dynamic* and *Deployment View*s. Several software components have been defined and connected in the *Static View* (see Figure 2.a). Among all system services we focus here on *MakePurchase* that is triggered whenever a customer wants to purchase a book or a movie[4]. The *Deployment View* (see Figure 2.b) shows the ECS allocation of software components on hardware nodes. Service requests from customers (client-side) pass through the *Internet*, and all the nodes in the server-side are connected by means of a 100 Mb/s *LAN*. Finally, for sake of simplicity we assume that both the client-side and the server-side are equipped with nodes having the same hardware characteristics for processors and disks.

We assume that a performance requirement has been defined on the *MakePurchase* service as follows: its average response time must not exceed 2 seconds under a workload of 150 customers.

Performance annotations

In order to analyze the ECS performance, several parameters must be defined:

(i) *Workload characterization*. A closed workload has been defined for the considered scenario, and the number of users for *MakePurchase* is 150, as stated in the requirement.

(ii) *Service demands definition*. Table III reports the average service demands (expressed in seconds) for the considered scenario. We suppose an average thinking time of 15 seconds for customers, whereas all the other service demands are obviously orders of magnitude lower than the thinking time.

TABLE III. ECS - *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

Performance analysis for the ECS

The performance analysis has been conducted by transforming the software model into a Queueing Network (QN) model [25], and by solving the latter with the JMT tool [26].

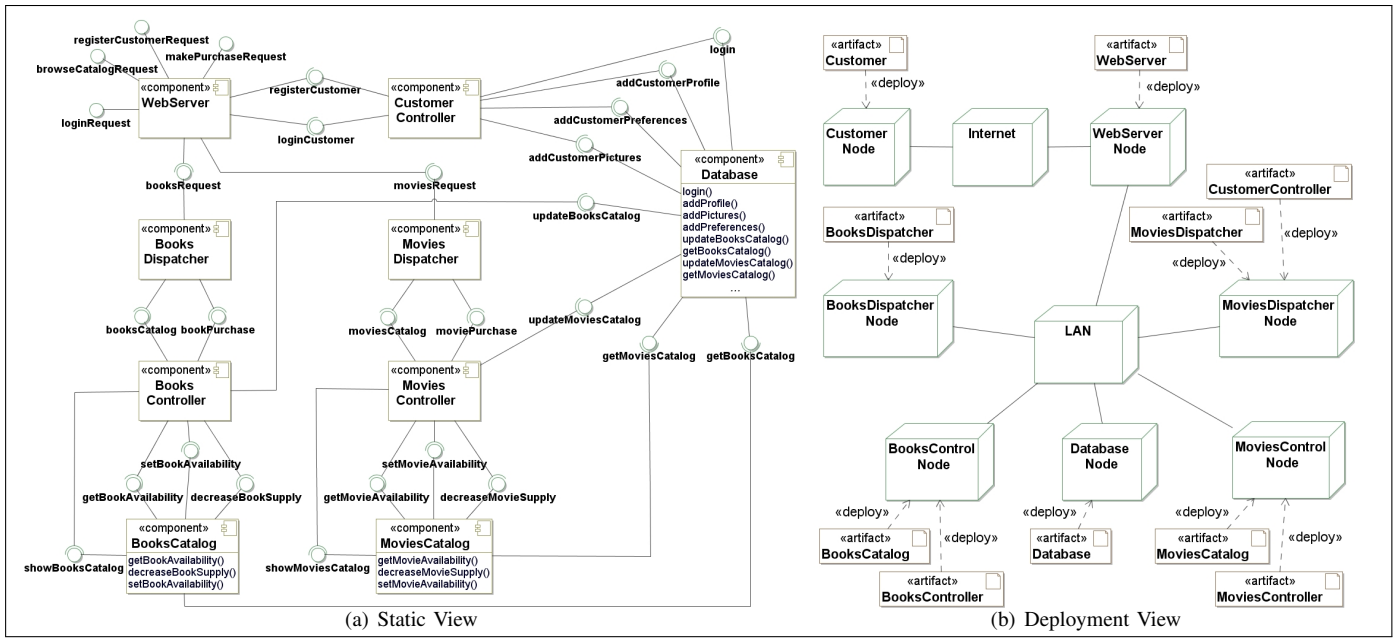
Table IV shows the resulting performance indices for the ECS software model; in particular, the average response times and utilizations of deployment nodes have been reported, as well as the average response time of the *MakePurchase* service.

TABLE IV. ECS - RESPONSE TIMES AND UTILIZATIONS FOR THE *MakePurchase* SERVICE.

	RT [sec]	U [%]
$MakePurchase$	17.16	-
$CustomerNode$	15	-
$WebServerNode$	0.017	13.11
$BooksDispatcherNode$	0.009	6.99
$MoviesDispatcherNode$	0.134	54.19
$BooksControlNode$	0.672	87.4
$MoviesControlNode$	0.934	91.77
$DatabaseNode$	0.396	78.66

As illustrated in Table IV the considered requirement is violated because, under a workload of 150 users purchasing a

[4]For sake of space, in Figure 2 we do not report the *Dynamic View* of the *MakePurchase* service. Readers interested to the dynamic view of ECS services may refer to [24].



(a) Static View (b) Deployment View

Fig. 2. ECS - (Annotated) Software Model.

product, the mean time elapsed (in the server-side) for each request (i.e. the average response time at the server-side) is 17.16 - 15 = 2.16 seconds, that is larger than the defined requirement of 2 seconds.

Antipatterns detection and refactoring for the ECS

For sake of simplicity, in the following we only focus on the Blob antipattern.

TABLE V. ECS - THRESHOLDS BINDING FOR THE BLOB ANTIPATTERN.

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

Table V reports some thresholds involved in the Blob antipattern specification. With these numerical values one instance of Blob is detected, i.e. the *MoviesController* component. Note that the *MoviesCatalog* component is not a Blob instance since 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 on which 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, the *BooksCatalog* component is not a Blob instance.

As a refactoring action *MoviesController* is redeployed on *MoviesDispatcherNode*. This leads to a response time of 1.98 seconds, that satisfies the requirement.

B. Emerging issues while varying thresholds

In this section we present an illustrative example whose experimental values for performance indices have been obtained while varying the considered thresholds. In particular, we vary $Th_{maxConnects}$ in the interval [4, 6] and $Th_{maxHwUtil}$ in the interval [85%, 95%], and we observe the influence of these variations on detection and refactoring activities.

TABLE VI. ECS - VARIATION OF BLOB THRESHOLDS.

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

Table VI summarizes the set of detected Blob instances while varying the numerical values of thresholds. The first column (i.e. #) identifies the variation. Furthermore, for each threshold the column *From* shows the initial value whereas the column *To* shows the value that the threshold assumes after the variation has been applied. Intuitively, the "–" symbol in the *To* column indicates that no variation has been made for the corresponding threshold value.

By increasing $Th_{maxConnects}$ from 5 to 6 and/or $Th_{maxHwUtil}$ from 90% to 95% no Blobs are detected. Since these thresholds represent upper bounds, it is evidently useless to explore further variations in this direction. On the contrary, while decreasing numerical values of thresholds we can notice that the number of detected antipatterns increases. In fact, by decreasing one or both of them (i.e. $Th_{maxConnects}$ from 5 to 4 and/or $Th_{maxHwUtil}$ from 90% to 85%) new Blob instances are detected in addition to the *MoviesController* component, up to the point that, with the variation #5, four Blob instances are detected.

For sake of this paper experimentation, we do not perform further decreases. In fact, it is not worth to indefinitely push

ahead the boundaries of thresholds variations, because false positive antipatterns might emerge. In order to establish fair boundaries more extensive experimentation is needed.

Blob instances have been refactored with the following actions:

- $redeploy$: this action moves the identified Blob component to the corresponding dispatcher node. For example, the redeployment of *MoviesCatalog* means that the component moves from *MoviesControlNode* to *MoviesDispatcherNode*. Such refactoring action is aimed at improving the utilization of the node where the Blob component was deployed. However, $Th_{maxHwUtil}$ refers to a performance index, hence we can not ensure a priori that the utilization of the latter node will be lower than $Th_{maxHwUtil}$ after the redeployment.

- $split$: this action equally distributes the connections of the identified Blob component between the latter one and several new components that are deployed on the corresponding dispatcher node. For example, the split of *MoviesCatalog* means that a new component is introduced in the software model and two connections have been moved to the new one. Such a refactoring action is aimed at reducing the number of connections of the Blob instance. Since $Th_{maxConnects}$ refers to a design feature, we can check by construction that the number of connections of the Blob component and the new one are lower than $Th_{maxConnects}$.

TABLE VII. ECS - VARIATION OF THE *MakePurchase* RESPONSE TIME ACROSS DIFFERENT BLOB REFACTORINGS.

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

Table VII summarizes the response times for the *MakePurchase* service while varying the threshold numerical values (i.e. #3, #4, and #5) and applying the $redeploy$ and $split$ refactoring actions[5]:

#3: After redeploying *MoviesCatalog* an average response time of 4.47 seconds is obtained. This is the worst case and it is due to the fact that 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.

By splitting *MoviesCatalog* an average response time of 1.66 seconds is obtained. This represents an improvement compared to the one deriving from splitting *MoviesController*. This is due to the fact that these two components have a different resource demand hence their operations' displacement lead to different demands for the *MoviesDispatcherNode*. 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.

#4: After redeploying *BooksController* an average response time of 1.84 seconds is obtained. This represents an improvement with respect to the one deriving from the first redeployment action. Anyhow, we can not guarantee that all the nodes involved in the redeployment action have an utilization higher than or equal to the modified $Th_{maxHwUtil}$ threshold (i.e. 85%), hence we need a further performance analysis step for the refactored model.

By splitting *BooksController* an average response time of 1.91 seconds is obtained. This does not represent an improvement compared to the one deriving from splitting *MoviesController*. Anyhow, 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 only have 2 connections.

#5: After redeploying *BooksCatalog* an average response time of 1.86 seconds is obtained, whereas by splitting it the average response time is 1.85 seconds. Both these refactoring actions lead to the same remarks of the previous action (i.e. #4).

Several observations regarding the antipatterns refactoring derive from our experimentation.

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 the removal of the antipattern instance. On the other hand, if a refactoring action refers to a threshold related to a performance index (e.g. hardware nodes utilization or throughput) we can not ensure that its application leads to the actual removal of the antipattern instance, in fact we need a further performance analysis step for the refactored model.

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

Finally, we must also consider possible legacy constraints that might restrict the set of applicable refactoring actions. In our example, let us suppose that *MoviesController* can not be refactored since it is a legacy component, hence the $split$ action on that component can not be applied anymore, and we can only apply the $redeploy$ action.

IV. CONCLUSION

In this paper we analyzed the influence of numerical thresholds on the capability of detecting and refactoring performance antipatterns. In particular we have shown on a simple example how the set of detected antipatterns instances may change while varying the threshold values, and we discussed the influence of thresholds on the complexity of refactoring actions.

It is certainly of great interest to extend the experiment reported here to other performance antipatterns. For this goal, more complex examples shall be considered. Instead, the extension of this work to different antipatterns that use thresholds in their definitions needs to be carefully planned, because domain-specific characteristics could be exploited.

[5]Note that performance indices have been obtained with the same model-based performance analysis presented before.

As future work we also intend to introduce confidence values that may be associated to antipattern instances to quantify the probability that the numerical threshold values support the actual antipattern presence. Furthermore, some fuzziness can be introduced for the evaluation of the threshold values [27] thus to make antipattern detection rules more flexible.

V. Acknowledgments

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).

References

[1] C. U. Smith, "Introduction to software performance engineering: Origins and outstanding problems," in *SFM*, ser. Lecture Notes in Computer Science, M. Bernardo and J. Hillston, Eds., vol. 4486. Springer, 2007, pp. 395–428.

[2] C. M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *FOSE*, L. C. Briand and A. L. Wolf, Eds., 2007, pp. 171–187.

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

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

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

[6] J. Xu, "Rule-based automatic software performance diagnosis and improvement," *Perform. Eval.*, vol. 69, no. 11, pp. 525–550, 2012.

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

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

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

[10] V. Cortellessa, M. De Sanctis, A. Di Marco, and 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.

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

[12] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and detection of soa antipatterns," in *International Conference on Service-Oriented Computing (ICSOC)*, 2012, pp. 1–16.

[13] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[14] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," in *Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 437–446.

[15] A. F. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *International Conference on Software Maintenance (ICSM)*, 2012, pp. 306–315.

[16] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 411–416.

[17] G. Travassos, F. Shull, M. Fredericks, and 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*, 1999, pp. 47–56.

[18] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[19] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 248–251.

[20] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[21] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. A. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[22] V. Cortellessa, A. Di Marco, and 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.

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

[24] ——, "E-Commerce System: use cases and dynamic view - Appendix," http://www.di.univaq.it/catia.trubiani/download/ECS-appendix.pdf, 2013.

[25] V. Cortellessa and R. Mirandola, "PRIMA-UML: a performance validation incremental methodology on early UML diagrams," *Sci. Comput. Program.*, vol. 44, no. 1, pp. 101–129, 2002.

[26] G. Casale and G. Serazzi, "Quantitative system evaluation with java modeling tools," in *WOSP/SIPEW International Conference on Performance Engineering(ICPE)*, 2011, pp. 449–454.

[27] S. S. So, S. D. Cha, and Y. R. Kwon, "Empirical evaluation of a fuzzy logic-based software quality prediction model," *Fuzzy Sets Syst.*, vol. 127, no. 2, pp. 199–208, Apr. 2002. [Online]. Available: http://dx.doi.org/10.1016/S0165-0114(01)00128-2