Digging into UML models to remove Performance Antipatterns

Vittorio Cortellessa, Antinisca Di Marco, Romina Eramo,
Alfonso Pierantonio, Catia Trubiani
Dipartimento di Informatica, University of L'Aquila
Via Vetoio,1 - 67010 Coppito
L'Aquila, Italy
{vittorio.cortellessa,antinisca.dimarco,romina.eramo,
alfonso.pierantonio,catia.trubiani}@univaq.it

ABSTRACT

Performance antipatterns have been informally defined and characterized as bad practices in software design that can originate performance problems. Such special type of patterns can involve static and dynamic aspects of software as well as deployment features. It has been shown that quite often the inability to meet performance requirements is due to the presence of antipatterns in the software design. However the problem of formally defining antipatterns and automatically detect them within a design model has not been investigated yet. In this paper we examine this problem within the UML context and show how performance antipatterns can be defined and detected in UML models by mean of OCL. A case study in UML annotated with the MARTE profile is presented where, after a performance analysis that shows unsatisfactory results, performance antipatterns are detected through an OCL engine. The identification of an antipattern suggests the architectural alternatives that can remove that specific problem. We show in our example that the removal of a certain antipattern actually allows to overcome a specific performance problem.

Categories and Subject Descriptors

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

General Terms

Performance, Antipatterns, Design.

Keywords

Unified Modeling Language, Object Constraint Language, Software Performance Engineering, Antipatterns.

1. INTRODUCTION

In the last decade the problem of automatically transforming software artifacts into performance models has been successfully tackled with a variety of approaches [6]. As opposite, the interpretation of the performance analysis results and the proposal of design alternatives to overcome performance problems is still based on the performance analysts' experience.

Figure 1 schematically represents the process executed, at a generic point of the software lifecycle, to assess and (if needed) improve the performance of a software system under development. Rounded boxes in the figure represent operational steps whereas square boxes represent input/output data. Vertical lines divide the process in three different phases: in the *modeling* phase a software model is built; in the *analysis* phase a performance model is obtained through model transformation, and such model is solved to obtain the performance indices of interest; in the *refactoring* phase the performance indices are interpreted and, if necessary, feedback is generated as refactoring actions on the original software model.

The modeling and analysis phases represent the forward path from an (annotated) software model to performance indices. In this path several approaches have been introduced for model transformation (see, for example, [6]) and well-founded performance model solvers have been developed (see, for example, [9]). Instead there is a clear lack of automation in the backward path that elaborates the analysis results and brings back to the software model some form of feedback. The refactoring phase in Figure 1, whose main task is the *result interpretation and feedback generation*, embraces the localization of performance flaws in the software model and their removal without violating design constraints[1]. Such activities are today exclusively based on the analysts' experience, and therefore their effectiveness often suffers the lack of automation.

Performance antipatterns represent a promising instrument to introduce automation in these activities. An antipattern is a well-known bad practice that should be avoided to achieve a better design. A performance antipattern identifies a practice that badly affects the software performance, and it may involve static and dynamic aspects of software as well as deployment features. A performance antipattern

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

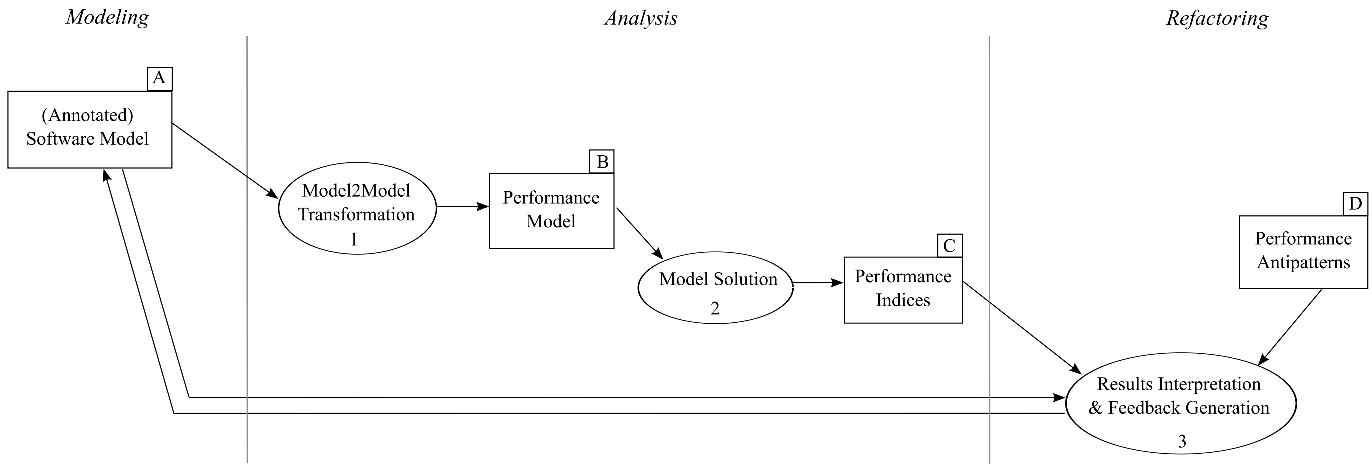


Figure 1: Software performance modeling and analysis process.

definition includes, beside the problem description, a possible solution of the problem. The main source of performance antipatterns is the work done over the last years by Smith and Williams [13] that have ultimately defined a number of 14 notation- and domain-independent antipatterns.

Few other papers present additional performance antipatterns defined across different technologies, but they are not as general as the ones defined by Smith and Williams.

The issue of detecting performance antipatterns has already been addressed in [11], where a rule-based performance diagnosis tool, named Performance Antipattern Detection (PAD), is presented. However PAD only deals with Component Based Enterprise Systems, targeting EJB applications. It is based on monitoring data from running systems, and it extracts the run-time system design and detects EJB antipatterns by applying rules to it. Therefore its scope is restricted to such domain, whereas in our approach the starting point is an UML model of the software system.

Another interesting work on the software performance diagnosis and improvements has been recently proposed in [14]: rules to identify patterns of interaction between resources are defined and specified as Jess rules [1]. Performance flaws are identified before the implementation of the software system, even if they are related only on bottlenecks (e.g. the "One-Lane Bridge" antipattern in Smith's classification) and long paths. However, performance issues are identified at the level of the LQN performance model, and the translation of these model properties into design changes could hide some possible refactoring solutions. Our approach refers both to performance and design features of the software system in the feedback generation process in order to maintain the whole information we need to choose the best design alternatives.

In this work we intend to support the idea that the localization and removal of performance flaws can greatly benefit from an automated detection of performance antipatterns [13]. Hence in Figure 1 the *Result Interpretation and Feedback Generation* step takes as input, beside the performance indices (label C) and the (annotated) software model (label A), the definition of performance antipatterns (label D). Performance antipatterns are searched in the annotated software model and a new software model is built by solving the found antipatterns as suggested in their definition itself.

The paper is organized as follows. Section 2 presents the problem statement and instantiates the software performance process of Figure 1 within the UML context; in Section 3 the textual definition of antipatterns (problem and solution) is introduced and then antipattern examples are represented as OCL rules; Section 4 describes the example that motivates the beneficial effects of detecting and solving performance antipatterns; finally in Section 5 we conclude the paper discussing the open issues related to the proposed approach and the future work.

2. PROBLEM STATEMENT

The search of performance flaws may be quite complex and needs to be smartly driven towards the problematic areas of the model. The complexity of this step stems from several factors: (i) performance indices are basically numbers associated to model entities and often they have to be jointly examined to let problems emerge; (ii) performance indices can be estimated at different levels of granularity and incomplete information often results from the model evaluation because it is unrealistic to keep under control all indices at all levels of abstraction; (iii) software models can be quite complex, they involve different characteristics of a software system (such as static structure, dynamic behavior, etc.), and performance problems sometimes appear only if those characteristics are cross-checked.

In this context antipatterns can play an important role since their definition includes performance indices, structural and dynamic features of the software system at different levels of abstraction. Therefore they represent a high potential mean to reduce the complexity of performance flaws identification and removal.

Although we aim at representing performance antipatterns as OCL rules, they are not UML-specific because each antipattern can be represented in other notations starting from its textual definition [13]. In fact in our previous work [7] we have introduced a technique based on first-order logic to specify system-independent rules that formalize known performance antipatterns. This property on one hand gives us the possibility to express a set of system properties under which an antipattern occurs with a certain degree of notation-independence. On the other hand, for the detection to be applied in practice, we need a software modeling notation (such as an ADL or UML itself) that can capture all defined system properties. In this paper the formalization of the antipatterns through OCL [2] rules is based on the well consolidated modeling UML notation [3] enhanced with MARTE profile [4].

In order to synthetically illustrate the context of this paper, each box of the software process in Figure 1 is instantiated in Table 1 [2].

General process	**This paper context**
(Annotated) Software Model	UML and MARTE profile
Model2Model Transformation	PRIMA-UML
Performance Model	Queueing Network
Model Solution	MVA
Performance Indices	Response time, Utilization, Throughput, ...
Performance Antipatterns	OCL rules
Results Interpretation & Feedback Generation	Detection and Solution of Antipatterns

Table 1: A customized overview of the process.

The starting point of the approach is a software system modeled with UML. *MARTE* profile provides all the information we need for reasoning on performance issues. The transformation from the software model to the performance model is performed with *PRIMA-UML*, i.e. a methodology that generates a Queueing Network model from UML diagrams [8]. Once the *Queueing Network* (QN) model is derived, classical QN solution techniques based on well-known methodologies [10] can be applied to solve the model, such as Mean Value Analysis (MVA). The performance model is analyzed to obtain the performance indices of interest: *response time*, *utilization*, *throughput*, etc. Performance antipatterns are defined as a set *OCL rules*, where each rule defines certain properties of the UML models and MARTE profile annotations. Such properties determine the occurrence of a performance antipattern.

The contribution of this paper is represented by the two bottom most entries of Table 1. Using OCL rules we dig into UML models to detect performance antipatterns. For each detected antipattern our approach suggests the designer a set of refactoring actions aiming at overcoming performance shortage.

3. OUR APPROACH

In this section we discuss our approach to detect and to solve performance antipatterns in UML. The specification of an antipattern includes i) the problem that represents a set of properties able to reveal performance issues as well as ii) the solution that represents a design alternative able to solve those performance problems. A performance antipattern specification is therefore fully described by *rules* representing the problem (see Section 3.1) and *actions* representing the solution (see Section 3.2).

Up to now, antipatterns have only been described in natural language, thus opening to various interpretations due to the natural language ambiguity. In this paper, we provide only one definition for an informal description reported in [13]. Of course, the rules and the actions that we propose reflect our interpretation of the natural language, whereas several other formalizations, all feasible at this stage of the research, of antipatterns can be originated as different interpretations of their informal descriptions. This unavoidable gap is an open issue in this domain, and certainly requires a wider investigation to consolidate the formal definition of performance antipatterns.

However, to reduce the impact of the natural language ambiguity, we have followed an example-driven approach in that we have analyzed several case studies in literature in order to capture (in different scenarios) the basic symptoms of each performance antipattern. Thereafter we have proposed our formalization basing on the (although limited!) acquired experience.

In the sequel of the section we discuss as an example the *Blob*, i.e. a performance antipattern fully explained in [12] whose informal definition is reported in Table 2. The same approach can be applied for the other antipatterns.

Antipattern	Problem	Solution
Blob	Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the applications data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the applications top-level classes, and to keep related data and behavior together.

Table 2: Blob performance antipattern.

The analysis of the antipattern *Blob* leads us to highlight that, from a performance perspective, a component creates problems by causing excessive message traffic. The communication is intensive because most business logics of the system is assigned to this component. The performance impact is clearly heavier on distributed systems, where the time needed to exchange data between two components is significant with respect to the computational time needed to perform operations. Hence it is necessary to consider additional hardware properties like the communication channels that inevitably affect the performance of the system. However in the centralized case some performance issues could arise due to the excessive load on the same CPU. Both (centralized and distributed) cases have been considered to give an extensive interpretation of the Blob antipattern.

An additional degree of freedom for antipattern definition is the level of abstraction. The entities on which the antipatterns can be defined change upon the abstraction of the system model. For example, the Blob antipattern can be easily reformulated by replacing components with classes if the antipattern search is performed at the design level instead of the architectural one. Maybe in other cases this porting is not so easy as for Blob.

3.1 Detecting antipatterns

From the informal representation of the *problem* a set of *rules* defined on UML and MARTE is built, where each rule addresses part of the antipattern problem specification. The rules are first described in a semi-formal natural language and then formalized by means of OCL queries.

In the following as aforementioned we discuss as an example the *Blob* antipattern whose informal problem definition

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

is reported in Table 2. The OCL query in Listing 1 detects the Blob antipattern by analyzing different rules described as follows. Each component in the defined context of the model is checked by means of the following rules in order to identify candidate Blob instances.

```
context Model ::
Blob() : Set(Component)

def: allComponents: Set(Component) =
     self.allOwnedElements() ->
     select(oclIsTypeOf(Component))
      .oclAsType(Component) -> asSet()

body : allComponents.usageRule()
              .interactionRule()
              .utilizationRule() -> asSet()
```

Listing 1: OCL query to detect the *Blob* antipattern.

Usage Rule: in a Component Diagram a complex controller component is surrounded by other components through many *usage* dependencies. The Usage Rule is implemented in the OCL query of Listing 2 that counts the number of interfaces each component uses and identifies all components whose number of interfaces is higher than the average number of interfaces used by all the other components.

```
context Component ::
usageRule() : Set(Component)

body : cc -> select(oclAsType(Usage) -> size()
       >= getComponentsUsageSize() /
          getComponentsSize())
```

Listing 2: OCL query for the *Usage Rule*.

Interaction Rule: in a Sequence Diagram there are lifelines that generate excessive message traffic. The Interaction Rule is implemented in the OCL query of Listing 3. For each component the number of messages sent is calculated. The result of the query is a set of components sending a number of messages higher than the average number of messages sent by all the components.

```
context Component ::
interactionRule() : Set(Component)

def: allLifelines: Set(Lifeline) =
     self.allOwnedElements()
     ->select(oclIsTypeOf(Lifeline))
       .oclAlType(Lifeline)->asSet()

body : self ->select (getAllComponentLifelines
       -> select(getSentMessages->size()
            > allLifelines.getSentMessages->size()
            / allLifelines->size())
      )
```

Listing 3: OCL query for the *Interaction Rule*.

Utilization Rule: according to our interpretation (see Section 3) in the following we consider two cases. The first case is the *centralized* one, i.e. the Blob component and the surrounding ones are deployed on the same processor. The performance issues due to the excessive load may come out by evaluating the CPU(s) device utilization, as shown in Listing 4 at lines 5-6. Note that the check to this goal is performed by extracting from the MARTE profile the *utilization* tagged value of the stereotype *GaExecHost*. The second is the *distributed* one, i.e. the Blob component and the surrounding ones are deployed on different processors. The performance issues due to the excessive message traffic may come out by evaluating the communication channel(s) device utilization, as shown in Listing 4 at lines 8-13. Note that the check to this goal is performed by extracting from the MARTE profile the *utilization* tagged value of the stereotype *GaCommHost*. The complete rule including both cases (i.e. centralized, distributed) is implemented in the OCL query of Listing 4.

```
1  context Component ::
2  utilizationRule() : Set(Component)
3
4  body : self ->select (
5      if singleDeployNode(self)
6      then    getOwningNode().utilization >= thr
7      else
8          getUsingComponent(self)
9          ->iterate (c: Component; result: boolean|
10         if getCommChannelNode(c).attribute.type
11                                    .include(self)
12         then getCommChannelNode(c).attribute
13                                .utilization >= thr
14       )
```

Listing 4: OCL query for the *Utilization Rule*.

Due to space limitation, auxiliary functions used in the query have been left out from this paper.

3.2 Solving antipatterns

From the informal representation of the *solution* a set of *actions* is built, where each action addresses part of the antipattern solution specification. The actions are described in a semi-formal natural language and proposed to designers for a manual selection of different options. We devised the following refactoring actions for the *Blob* antipattern:

Business Logics Action: delegate some business logics from the Blob component to the other components that were used only as data containers thus to reduce the number of messages exchanged and to keep data and behavior together in the same component.

Redeployment Action: if the Blob component and the surrounding ones are distributed on different hardware machines than the lower number of messages should automatically improve the utilization of the network, otherwise check if it is possible to distribute more uniformly software components on the hardware platforms.

Currently we do not automate the removal of antipatterns but it will be matter of future works.

4. EXPERIMENTATION

In this section, we report the experimentation of our approach on an e-commerce example that motivates the beneficial effects of detecting and removing performance antipatterns in a UML model. In particular, this example shows how the OCL rules, introduced in Section 3.1, have been used to detect antipatterns and, consequently, how effective are the refactoring actions introduced in Section 3.2 to improve the system performance. The refactoring has been manually performed whereas the detection is automated by the OCL rules.

The experiment has been conducted as follows. Starting from an E-Commerce System (ECS) modeled in UML profiled with MARTE, we have generated a QN model that, once solved, revealed problems in the system since the performance indices did not fulfill the requirements. Then the OCL engine has detected some performance antipatterns in the model. The solution of one antipattern suggests how to obtain a new system model. Such model undergoes the same process as the original one and shows better performance results. This process can be iterated several times until the performance requirements are satisfied.

ECS is a web-based system that manages business data: customers browse catalogs and make selections of items that need to be purchased. At the same time, agents can upload their catalogs, change the prices, the availability of products etc. ECS offers several services to users, such as browsing catalogs and making purchases. The former can be perfomance-critical because it is required by a large number of (registered and not registered) customers, whereas the latter can be perfomance-critical because it requires several database accesses that can drop the system performance.

For sake of space in the following we only consider the *browseCatalog* service that is invoked with a probability of 0.98. The performance requirement we consider is that this service must be completed (in average) in less than 1.5 seconds when the total workload of 150 requests/second occurs in the system.

We adopt the Prima-UML methodology [8] in the forward path from an (annotated) software model to a QN performance model (i.e. the operational step *Model2Model Transformation* of Figure 1). PrimaUML requires as inputs: an Use Case Diagram annotated with the operational profile, Sequence Diagrams annotated with the workload and the resource demands, and a Deployment Diagram annotated with the characteristics of the platform devices. The UML Deployment Diagram for ECS is shown in Figure 4. The diagram is annotated with the characteristics of the hardware nodes through MARTE stereotypes, for example to specify the CPU characteristics (i.e. speedFactor and schedPolicy tags) and the network bandwidth (i.e. blockT tag).

	Service Demand (input parameters) ECS	Service Demand (input parameters) $ECS \setminus \{Blob\}$
wan	1040 msec	1040 msec
lan	**396** msec	**242** msec
$webServerNode$	4 msec	4 msec
$libraryNode$	**9** msec	**8** msec
$controlNode$	**6** msec	**7** msec
$databaseNode_{cpu}$	15 msec	15 msec
$databaseNode_{disk}$	30 msec	30 msec

Table 3: Input parameters of the ECS and $ECS \setminus \{Blob\}$ queueing network models.

The performance indices of the ECS system are obtained by solving the QN obtained through the transformation step with the WinPEPSY-QNS tool [5]. The input parameters adopted for this experiment are reported in Table 3, where the first column represents the parameters of the original model and the second column the ones of the refactored model after the antipattern solution. For the original model the first row of Table 3 shows that the *wan* requires a service demand of 1040 msec. This value is obtained considering that the resource has a maximum bandwidth of 208 msec/KByte (as shown in Figure 4) and in the *browseCatalog* scenario there are 4 messages crossing this network (i.e. messages labeled with numbers 1, 2, 27, 29 in Figure 4). As annotated in Figure 4 the total size of those messages is 5 KBytes that gives 208*5 = 1040 msec.

The performance analysis of the ECS original model reveals that the response time of the *browseCatalog* service (under a workload of 150 requests/second) is 1.61 seconds, so it does not meet the required 1.5 seconds. Since the requirement is not satisfied we apply our approach to detect performance antipatterns.

The OCL rule-engine application detects three antipatterns that are fully described in Table 4. Due to space limitation but without loss of generality, in the following we only discuss the solution of the *Blob* antipattern, although different strategies can be adopted to (more or less quickly) converge towards a solution of the performance problem. However, this is matter of our future work. According to the antipattern solution proposed in Table 4, the software component *libraryController* shall not be the intermediate component for services provided by *bookLibrary* component. We refactor the model and we obtain a new model named $ECS \setminus \{Blob\}$ where the *Blob* antipattern is solved [3].

In Figure 3 we illustrate the effect of the refactoring actions on the static architecture of ECS, here represented as a UML component diagram. Indeed, Figure 3(a) represents the original structure of the ECS architecture where we highlight, in the shaded box, the portion of the architecture that might evidence the Blob antipattern presence. In more details, the *libraryController* component requires excessive processing resources (as indicated by the utilization of the node it is deployed on) and generates excessive message traffic by exploiting its usage dependencies with the *bookLibrary* component. Figure 3(b) shows how the shaded part has been refactored to remove the Blob antipattern. The refactoring consists in re-designing two components and the connector between them in order to guarantee the balance of the workload between those two components.

However, the model has to be refactored while keeping related data and behavior consistent. In fact, as illustrated in [7], an antipattern can concurrently affect static, dynamic and deployment characteristics of a model. As a consequence, in the $ECS \setminus \{Blob\}$ system components are refactored as *libraryController'* and *bookLibrary'* components, but also the *manageBook'* interface behavior has to be redesigned in order to manage in a different way the inner operations of the component it belongs.

In Figure 4 we report the Sequence Diagram of the ECS system and the refactoring actions that lead to $ECS \setminus \{Blob\}$. In particular, the shaded boxes highlight the *Blob* antipattern problem: the *libraryController* component generates excessive message traffic. Shaded boxes are replaced according to the *Blob* antipattern solution that is: the *libraryController* component delegates the management of data to the *bookLibrary* component.

The input parameters of the $ECS \setminus \{Blob\}$ QN model are reported in the second column of Table 3. Bold entries of

[3]We adopt the following notation: $System \setminus \{Antipat_x, \ldots, Antipat_N\}$ to denote that the initial system has been refactored by applying the solution of the antipatterns specified between brackets. It is worth to notice that the solution of an antipattern is not deterministically proved to solve performance issues thus in the refactored system we do not exclude that new antipatterns might emerge.

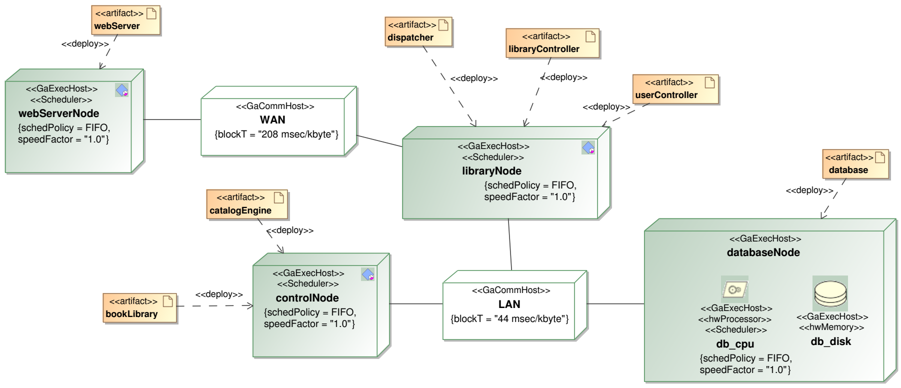
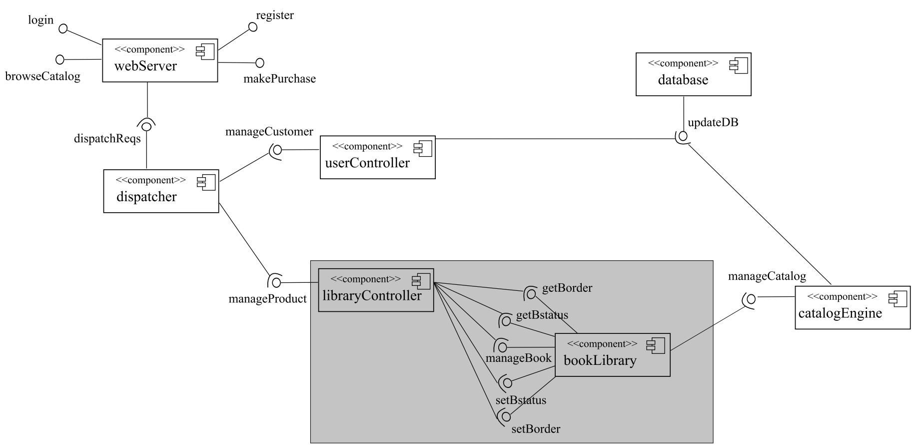


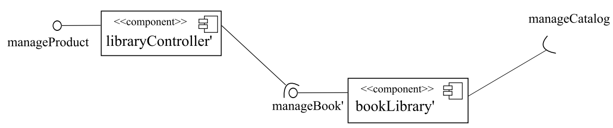
Figure 2: UML Deployment Diagram of the ECS system.

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

Table 4: ECS Detected Performance Antipatterns.



(a) "Blob" problem



(b) "Blob" solution

Figure 3: UML component diagram of the ECS and $ECS \setminus \{Blob\}$ systems.

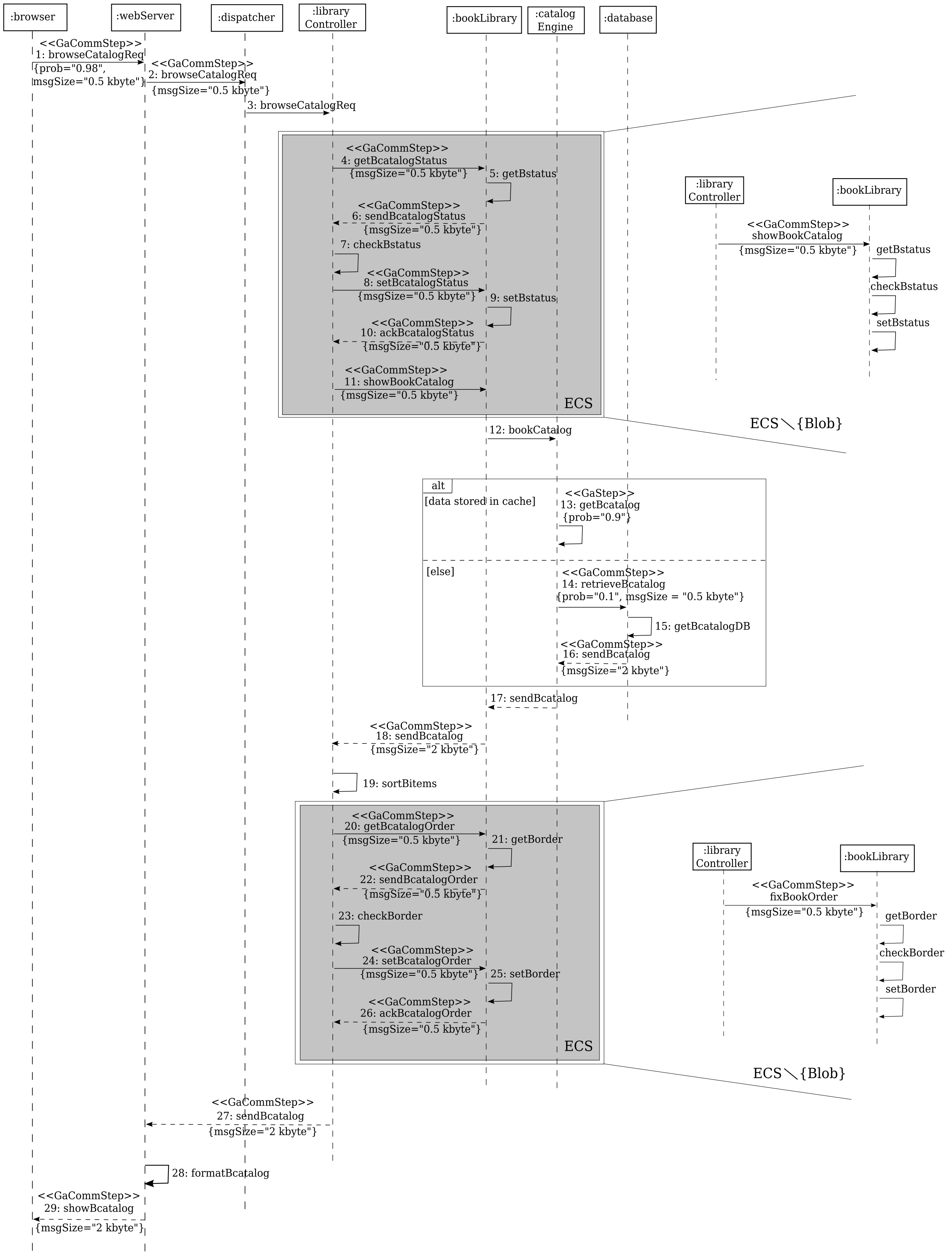


Figure 4: UML sequence diagram of the *browseCatalog* service in the ECS and $ECS \setminus \{Blob\}$ system models.

Table 3 highlight the values changing across the two different architectures of the system (i.e., ECS and $ECS \setminus \{Blob\}$). For example, the second row shows that the *lan* requires a service demand of 396 msec for the ECS system. Such resource has a maximum bandwidth of 44 msec/KByte (as shown in Figure 4). In the ECS system the *browseCatalog* scenario has 12 messages circulating on this network (i.e. messages labeled with numbers 4, 6, ..., 24, 26 in Figure 4) and, as annotated in Figure 4, the total size of those messages is 9KBytes, thus it gives 44*9 = 396 msec. In the $ECS \setminus \{Blob\}$ system the *browseCatalog* scenario has a reduced number of messages over this network thus the total size of messages exchanged is 5.5 KBytes that gives 44*5.5 = 242 msec, as shown in the second column of Table 3.

PrimaUML is again applied on the $ECS \setminus \{Blob\}$ system. The performance results of the corresponding queueing model reveal that the response time of the *browseCatalog* service is 1.44 seconds under a workload of 150 requests/second, thus the requirement is satisfied.

Requirement	Required Value	Observed Value	
		ECS	$ECS \setminus \{Blob\}$
RT($browseCatalog$)	1.5 sec	1.61 sec	1.44 sec

Table 5: Response time of $browseCatalog$ service.

The performance index of interest is summarized in Table 5. In this case the *Blob* antipattern has provided a relevant information because its removal improves the response time so that the requirement can be fulfilled.

5. CONCLUSION

In this paper we have presented an approach, based on antipatterns, that aims at identifying performance problems in UML models and removing them. The antipattern detection is based on OCL rules that formalize the informal existing definitions of performance antipatterns.

We have reported here preliminary results of an experiment that falls within a wider study we are conducting on the interpretation of performance results and generation of architectural feedback. This experiment allowed us to ground our general approach to a widely used notation, like UML. The results obtained through OCL rules are promising, although several open issues remain to be addressed within the UML context as well as in a more general vision.

With regard to UML a key question is whether OCL is a powerful-enough language to unambiguously represent all known performance antipatterns. We have modeled only few of them, but it would be interesting to extend such representation to other performance antipatterns. In general, there is a gap between the informal description of an antipattern (as a problem) and its formal representation, whatever the adopted notation is. Several formalizations could be equivalently representing the same antipattern, so a deeper study is needed to find (possibly multiple) unambiguous representations of antipatterns. This would allow a sharper ability of detecting antipatterns.

The antipattern solution (i.e. the model refactoring) that in this paper has been manually executed, opens to multiple problems to be tackled. For sake of space, here we like only to recall that, once a number of performance antipatterns are detected, a certain strategy has to be introduced to decide which ones have to be solved in order to quickly converge towards an acceptable improvement of system performance. Such a strategy can be based on different factors that can be architectural ones (e.g. legacy constraints that do not allow to solve a certain antipattern, or incompatibility between solutions of different antipatterns) or non-functional ones (e.g. an antipattern solution is too expensive or it can badly affect the software reliability).

We are facing the above issues in order to build a general framework for antipattern detection and solution based on model-driven techniques and independent of any modeling notation.

6. ACKNOWLEDGMENTS

This work has been partly supported by the italian project PACO (Performability- Aware Computing: Logics, Models, and Languages) funded by MIUR.

7. REFERENCES

[1] JESS, the Rule Engine for the Java Platform, *http://www.jessrules.com/jess/index.shtml.*

[2] OCL 2.0 Specification, OMG, *http://www.omg.org/cgi-bin/doc?formal/06-05-01.*

[3] UML 2.0 Superstructure Specification, OMG, *http://www.omg.org/cgi-bin/doc?formal/05-07-04.*

[4] UML Profile for MARTE, OMG, *http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf.*

[5] WinPEPSY-QNS, a tool for calculating performance measures of queueing networks, *http://www7.informatik.unierlangen.de/prbazan/pepsy.*

[6] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[7] V. Cortellessa, A. Di Marco, and C. Trubiani. Performance antipatterns as logical predicates. In *IEEE International Conference on Engineering of Complex Computer Systems*, 2010. to appear.

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

[9] C. Hirel, R. Sahner, X. Zang, and K. Trivedi. Reliability and Performability Modeling using SHARPE 2000. In *Computer Performance Evaluation: modelling techniques and tools*, volume 1786 of *LNCS*, pages 345–349, 2000.

[10] R. Jain. *Art of Computer Systems Performance Analysis.* 1991.

[11] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–90, 2008.

[12] C. U. Smith and L. G. Williams. Software performance antipatterns. In *2nd International Workshop on Software and Performance*, 2000.

[13] C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, 2003.

[14] J. Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP'08, Workshop on Software Performance*, pages 1–12, 2008.