Antipattern-Based Model Refactoring for Software Performance Improvement

Davide Arcelli, Vittorio Cortellessa, Catia Trubiani
Università degli Studi dell’Aquila, 67100 L’Aquila, Italy
{davide.arcelli, vittorio.cortellessa, catia.trubiani}@univaq.it

ABSTRACT

Identifying and removing the causes of poor performance in software systems are complex problems due to a variety of factors to take into account. Nowadays these problems are usually tackled after the software deployment only with human-based means, which frequently boil down to developer skills and previous experiences. Performance antipatterns can be used to cope with these problems since they capture typical design patterns that are known leading to performance problems, as well as refactoring actions that can be taken to remove them.

The goal of this paper is to introduce an approach that allows the refactoring of architectural models, based on antipatterns, that aims at providing performance improvement. To this end, we use a Role-Based Modeling Language to represent: (i) antipattern problems as Source Role Models (SRMs), and (ii) antipattern solutions as Target Role Models (TRMs). Hence, SRM-TRM pairs represent new instruments in the hands of developers to achieve architectural model refactorings aimed at removing sources of performance problems. Model refactoring for antipattern removal can be in fact obtained by replacing an SRM with the corresponding TRM. This approach has been applied to a case study in the e-commerce domain, whose experimental results demonstrate its effectiveness.

Keywords
Software Performance, Model Refactoring, Performance Antipatterns, Roles

1. INTRODUCTION

Identifying and removing the causes of poor performance in software systems are complex problems due to a variety of factors to take into account. Similarly to other non-functional properties, performance is an emergent attribute of software, as it is the result of interactions among software components, underlying platforms, users and contexts [26]. The current approaches to these problems are mostly based on the skills and experience of software developers or, in the best cases, of performance analysts. Quite sophisticated profiling tools have been introduced to monitor performance of running applications [20], but it is well-known that the costs of solving performance problems at runtime is orders of magnitude larger than the ones at early phases of the software lifecycle [14]. Therefore instruments that help to identify and remove causes of software performance problems early in the lifecycle are very beneficial.

In the last two decades the concept of performance antipattern has been used for “codifying” the knowledge and experience of analysts. Smith and Williams have been the most prolific researchers that, based on their practical experience, have ultimately specified 14 performance antipatterns [22]. A performance antipattern identifies a problem, i.e. a bad practice that negatively affects the software performance, and a solution, i.e. a set of refactoring actions that can be carried out to remove it.

We have based our recent research work on this repository of knowledge with the aim of making it a cornerstone in the process of identification and removal of performance problems. Since the antipatterns had been originally defined in natural language, we have first tackled the problem of providing a less ambiguous representation [7]. However, performance antipatterns are very complex (as compared to other software patterns) because they are founded on different characteristics of a software system, spanning from static to behavioral to deployment, and they additionally include values of performance indices. This high complexity requires multi-view representations, as demonstrated in [7].

Thereafter we have introduced several techniques aimed at detecting performance antipatters in software architectural models [6, 25].

In this paper we move a further step ahead, in that we undertake the problem of removing performance antipatterns detected in an architectural model. The goal of this paper is to introduce a model-based approach that allows to formalize the refactoring embedded into performance antipattern definitions. Our approach enables developers to focus on (potential) sources of performance problems and suggests how to refactor models in order to remove problems.

Figure 1 summarizes the process envisaged to address this issue and the context where it works. Rounded boxes represent operational steps whereas square boxes represent input/output data. The primary input to the process is an architectural model annotated with performance-related information, such as workload, service demands, and/or hard-
ware characteristics\(^{1}\). Basing on performance indices obtained from its analysis, and on the definitions of performance antipatterns, the performance antipatterns detection step produces a list of antipatterns occurring in the software architectural model\(^{2}\).

![Diagram of Antipatterns-based process for the detection and solution of performance problems.](image)

Shaded boxes of Figure 1 represent the focus of this paper. The antipattern removal process starts with a Refactoring-Based Annotation step. This is necessary because the detection only results in the model fragment that represents the antipattern, whereas other contextual information may be needed to perform refactoring actions (see Section 2). Hence, a Refactoring-Annnotated model contains all the information useful to remove the detected antipatterns. A solution step can then be executed on the latter model, and a refactored architectural model is the result of the whole process.

We have used the Role-Based Modeling Language (RBML) [12] to represent: (i) antipattern problems as Source Role Models (SRMs), and (ii) antipattern solutions as Target Role Models (TRMs). Model refactoring aimed at removing antipatterns consists in replacing one or more SRMs with their corresponding TRMs. Hence, SRM-TRM pairs represent new instruments in the hands of developers to support architectural model refactoring aimed at identifying and removing sources of performance problems.

In this paper the SRM-TRM replacement is not automated. However, we have already experimented that the formalization of a SRM-TRM pair, as proposed here, is an excellent starting point for the generation of the underlying transformation from SRM to the corresponding TRM. This transformation will represent the refactoring actions that the solution step executes to remove antipatterns, and this is part of our future work.

The contribution of this paper is not limited to the definition of several SRM-TRM pairs for existing antipatterns, but we show that RBML, that was used in [12] only for design patterns, can be nicely adopted to define refactoring actions even for performance antipatterns to come. The benefit of using RBML is that the concept of role is very suitable to capture the heterogeneity of the knowledge (i.e. architectural model properties, performance indices, etc.) underlying the specification of antipatterns !!!—<<<\(^{3}\), >>>—!!!!.

The paper is organized as follows: Section 2 provides background through an example antipattern; Section 3 describes the main contributions of this paper: (i) the usage of a Role-Based Modeling Language, and the specification of (ii) a set of Source Role Models (SRMs) for existing performance antipatterns, and (iii) a set of corresponding Target Role Models; Section 4 shows the approach at work on an e-commerce case study; Section 5 discusses the relevant issues of the proposed approach; Section 6 presents related work, and finally Section 7 concludes the paper and provides directions for future research.

## 2. AN EXAMPLE ANTIPATTERN

The aim of this section is to provide background on performance antipatterns through the description of one example, i.e. the Empty Semi Trucks (EST) [22].

Table 1 reports the textual description of the EST antipattern, as originally defined in literature [22]. In the following we propose a graphical representation of this antipattern in a UML-like notation. Note that such representation reflects our interpretation of the textual description, and it is conceived to capture one reasonable illustration of both the antipattern problem and solution, but it does not claim to be exhaustive\(^{4}\).

<table>
<thead>
<tr>
<th>Antipattern</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty Semi Trucks</td>
<td>Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.</td>
<td>The Batching performance pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.</td>
</tr>
</tbody>
</table>

Table 1: Textual description of a performance antipattern.

As reported in Table 1, the EST antipattern may be due to inefficient use of available bandwidth, an inefficient interface, or both. Solution actions are proposed in the rightmost column for all these cases. In this section we only consider the case of inefficient interface, and we assume that the solution makes use of the Session Facade design pattern [23].

Figure 2(a) describes an annotated software architectural model \(M\) with an Empty Semi Trucks occurrence. \(M\) is a

\(^{1}\)This information is used to transform software models (e.g. UML models) into performance ones (e.g. Queueing Networks), thus enabling the analysis that carries out performance indices of interest (e.g. utilization) [9].

\(^{2}\)We do not provide details on the antipattern detection step because it is not the focus of this paper; however readers interested can refer to our previous work [7, 6].

\(^{3}\)Note that we cannot use RBML as-is for our purposes, so we had to heavily modify it, as illustrated in [7].

\(^{4}\)Readers can refer to [7] for a discussion on the gap between informal and formal representation of performance antipatterns.
multi-view model because, as discussed in Section 1, a performance antipattern combines characteristics of software systems that span over different views (i.e., static, dynamic, deployment).

Let us assume that a requirement is defined on the utilization of platform devices [15], i.e., they have not to be used more than 80%. Figure 2(a) shows the violation of this requirement for the processing node PN₁ and the network link Net, whose utilization values are, respectively, 85% and 92%.

According to the vocabulary and the detection rules defined in [7], an EST antipattern is detected in this model because there is a software entity instance (SwRx) that: (i) generates an excessive message traffic towards another software entity instance (RemInstance), (ii) is deployed on a processing node with a high utilization value, i.e. PN₁ (0.85), and (iii) the network link on which the message traffic is generated shows a high utilization value (0.92). The simultaneous occurrence of such properties leads to assess that the SwRx instance originates an EST antipattern.

Note that detection rules [7] contain thresholds that basically represent systems features, in particular they may refer to upper/lower bounds for: (i) design properties (e.g. excessive message traffic); (ii) performance results (e.g. high, low utilization). Numerical values can be assigned by software architects basing on heuristic evaluations, or they can be obtained by monitoring the system (see more details in Section 4).

Figure 2(b) describes a refactored model M’ that results from applying some of the changes defined in the EST solution, in particular: the communication between SwRx and RemInstance must be restructured in order to reduce the number of messages sent over the network. Hence, the Session Facade design pattern [23] is applied: two software entity instances are introduced (i.e., RemoteFacade and LocalFacade) in the architectural model, and the local communication replaces the remote one.

As consequences of the previous action, less data are exchanged across a communication device and the utilization of the processing node hosting SwRx improves (to 0.63) as well as the utilization of the network link (to 0.5). Note that utilization values of these latter devices improve despite PN₂, whose utilization grows from 0.67 to 0.75. However, the requirement is satisfied in the model M’ since all devices have utilization values lower than the 0.80 required one.

3. MODEL REFACTOING APPROACH

In this section we provide a description of our approach summarized in the shaded boxes of Figure 1. In particular, we motivate the usage of the Role-Based Modeling Language (RBML) to represent performance solutions through the specification of: (i) a set of Source Role Models (SRMs) identifying the most critical elements in a software architectural model (see Section 3.1); (ii) a set of Target Role Models (TRMs) specifying the refactoring that must be applied to the architectural model (see Section 3.2).

The complexity of architectural models refactoring stems from several factors, and in particular the fact that a refactoring action is rarely context-free. For example, consider that a refactoring is meant to split a software component in two components and re-deploy one of them. This change in the static view of the model may cause the necessity to investigate other features of the system, e.g., the deployment...
view, in order to determine the most suitable deployment node to host the new component (i.e., very likely the less utilized one). Hence, context information can be fundamental for refactoring representation.

Figure 3 shows the RBML metamodel [12] and the relationship that we have defined with antipattern specifications. The shaded boxes represent the main concepts of RBML, that are: Role Model, Source and Target Role Models, Role and Realization Multiplicity.

![Diagram](image)

**Figure 3: Role-Based Modeling Language applied to performance antipatterns.**

*Role Models* rest on the antipattern specifications that have been introduced in [5], where we defined a Performance Antipattern Modeling Language (PAML) on the basis of [7]. *Source and Target Role Models* are used to extend the semantics of the role model and represent, in our case, a performance problem and its refactoring respectively. In order to solve a performance antipattern several actions can be taken, hence several Role Models (i.e. SRM-TRM pairs) can be associated to the same antipattern. A Role Model contains a set of *Roles*, where each role represents an architectural model element involved in the corresponding antipattern specification. Each role has a *Realization Multiplicity* that specifies the number of its realizations, i.e. the role instances.

For sake of space this section only reports one example of Source and Target Role Models for the EST antipattern. However we have used RBML to build a large set of SRM-TRM pairs. Our repository of Role Models [1] currently contains: three SRM-TRM pairs for the Blob antipattern, one SRM-TRM pair for the Concurrent Processing Systems antipattern, and two SRM-TRM pairs for the Pipe and Filter antipattern [22].

### 3.1 Source Role Models (SRMs)

Figure 4 shows a Source Role Model (SRM) for the EST antipattern.

The left dashed box of Figure 4 contains the output of the antipattern detection, where a software entity instance *SwRx* has been identified, in a service *ServiceS*, as an instance of the EST antipattern. Note that model element roles (e.g. *SoftwareEntityInstance, Service, Behavior*) have been inherited from the Antipattern Specification (see Figure 3) we defined in [5], whereas SRM classes (e.g. *SwRx, ServiceS, ServiceSBehavior*) define the model elements these roles apply to. For instance, *SwRx* plays the sender role in EST (as illustrated in Figure 2(a)), because it sends an excessive amount of remote messages named *RemInstanceOpCall* (i.e. *callOperation()* in Figure 2(a)). EST is detected because the number of these messages is higher than *MaxRemMsgs*, namely an upper bound threshold.

As outlined above, the detection step captures, along with the software entity instance, only the other model elements that are necessary to identify the bad practice, whereas no information on the context is reported. Context information is instead necessary to show up the additional model elements that can support the removal of the bad practice. Such information is introduced by the designer in the Refactoring-Based Annotation step (see Figure 1).

The right dashed box of Figure 4 contains the context information we need to define the refactoring actions for the EST antipattern in this pair of role models. *ServiceBehavior* represents the scenario where the bad practice takes place. Such behavior contains three types of messages: (i) *RemInstanceOpCall* corresponds to the set of remote messages sent to call a remote operation, as captured by the detection; (ii) *RemInstanceOpCallReply* corresponds to the set of remote messages replying to the call (i.e. *reply()* in Figure 2(a)); (iii) *InstancedOpExec* corresponds to the set of self-messages invoking a certain operation (i.e. *Op()* in Figure 2(a)). *SwRx* communicates with a remote software entity instance annotated as *RemInstance* (as illustrated in Figure 2(a)). Context also provides information on the deployment of interacting entities. In fact, they can be considered as remote because they are deployed on different processing nodes, i.e. *SwRxDeployNode* and *RemInstDeployNode* in Figure 4 (*PN*_1 and *PN*_2 in Figure 2(a)).

### 3.2 Target Role Models (TRMs)

A Target Role Model (TRM) is the result of refactoring actions applied to the corresponding SRM. Three types of refactoring actions are devised in our approach: (i) *keep-as-is*, i.e. a role model element and/or a relationship is not modified; (ii) *delete*, i.e. a role that appears in the SRM is deleted in the TRM (e.g. splitting a software entity instance leads to delete it and replace it with new ones); (iii) *add*, i.e. the TRM contains a new role that does not appear in the SRM (e.g. duplicating a software entity instance).

Figure 5 shows the Target Role Model (TRM) for the EST antipattern. The dashed box contains the model elements introduced for representing the refactoring, with the exception of *MaxRemMsgs* that, however, in SRM was related to a different element (i.e. from *RemInstanceOpCall* to *LocalFacadeOpCall*).

In particular, two software entity instances have been added, i.e. *RemoteFacade* and *LocalFacade*, and the communication has been restructured as follows. Two types of messages have been added for each new software instance: (i) *RemFacadeOpCall* and *RemFacadeOpCallReply* have been added for *RemoteFacade*; (ii) *LocalFacadeOpCall* and *LocalFacadeOpCallReply* have been added for *LocalFacade*. Note that *RemoteFacade* plays here the sender role for *LocalFacadeOpCall* messages (i.e. *requestData* messages in Figure 2(b)), whose receiver role is played by *LocalFacade*. The role *role1..* that specifies the number of its realizations, i.e. the role instances.

5For example, in Figure 2(a) the *SwRx* software entity instance plays a crucial role for identifying the EST problem. Such role indeed appears in the corresponding SRM of EST, as ≫SoftwareEntityInstance Role ≫ of *SwRx* in Figure 4. 7Note that the thresholds values of antipatterns are verified by means of OCL expressions, please refer to [1].
Figure 4: A Source Role Model for the Empty Semi Trucks antipattern.

Figure 5: A Target Role Model for the Empty Semi Trucks antipattern.
multiplicity of remote calls is here lower than MaxRemMsgs, due to the application of the Session Facade design pattern as illustrated in Figure 2(b).

Note that a SRM-TRM pair may not be sufficiently expressive to define the refactoring actions, because behavioral details may be needed to illustrate modifications of interaction patterns. To this end, we use Interaction Role Models (IRMs) as support for Source and Target ones. An IRM is aimed at specifying the refactoring of behavioral patterns, like the changes in the communication among software instances to avoid an unbalanced amount of messages [1], >>>—!!!!![8]

4. CASE STUDY

In this section we present a case study in the e-commerce domain to show the application of our antipattern-based approach, as well as the performance improvements induced on the software architectural model.

We first describe the (annotated) software architectural model of the E-Commerce System (ECS), then the process in Figure 1 is stepwise applied. Finally, the results obtained from the experimentation are discussed.

Figure 6 shows the ECS annotated model, that is a web-based system that manages business data. In particular, Figure 6(a) describes the static view of the system (i.e. the software entity instances and their relationships), whereas Figure 6(b) shows the deployment view (i.e. the allocation of software entity instances on processing nodes). Among all provided services, we focus in this paper on visitors and customers that browse catalogs (i.e. BrowseCatalog service), and customers that select items to purchase (i.e. MakePurchase service).

Performance requirements have been defined on the response times of the BrowseCatalog and MakePurchase services (under a maximum workload of 200 requests/sec) as follows: the BrowseCatalog service has to be completed in 1.2 seconds, and the MakePurchase service in 2.5 seconds.

The performance analysis has been conducted by transforming the software architectural model into a Queueing Network (QN) performance model [9], and by solving the QN model with the JNIT tool [3].

Table 2: Response time of the ECS model.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Required Value</th>
<th>Predicted Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT(BrowseCatalog)</td>
<td>1.2 sec</td>
<td>1.4 sec</td>
</tr>
<tr>
<td>RT(MakePurchase)</td>
<td>2.5 sec</td>
<td>2.69 sec</td>
</tr>
</tbody>
</table>

Table 2 reports the response times of the ECS model. First column reports the required index, second column the required value, and third column the corresponding predicted value (as obtained from the QN analysis). As it can be noticed both services have response times that do not fulfill the required ones.

Hence we apply our approach to detect antipatterns [7] that reports the occurrence of four performance antipatterns: Blob, Concurrent Processing Systems (CPS), Empty Semi Trucks (EST), and Pipe and Filter (P&F). For sake of space, in the following we only discuss the EST antipattern.

Figure 7 shows the (Refactoring-annotated) Software Architectural Model. The Source Role Model for the EST antipattern (see Figure 4) has been applied: UserController is the software entity instance, identified as SubRx in the ECS architectural model, that originates the EST antipattern, since it sends to the Database instance, identified as RemInstance in the ECS architectural model, more than 20 (as specified in the ThresholdSpecification) remote messages; the communicating instances are remotely deployed on LibraryNode and DatabaseNode, respectively.

Figure 8 shows the refactored ECS model, where the Target Role Model for the EST antipattern (see Figure 5) has been applied. In agreement with the TRM: (i) two new software entity instances have been added, i.e. RemoteFacade and LocalFacade (see Figure 8(a)); (ii) the communication has been refactored and local messages have replaced remote ones (see Figure 8(b)); (iii) the software entity instances RemoteFacade and LocalFacade have been respectively deployed on LibraryNode and DatabaseNode (see Figure 8(c)).

Up to this point we have shown on the ECS case study the application of our approach to model and solve performance antipatterns in architectural models, which is the main goal of this paper. Since this approach allows to easily produce refactored models, we have used it to analyze the sensitivity of the ECS performance to the removal of different antipatterns. This analysis is aimed at showing the new capabilities that analysts have in their hands when using our approach. However, although the removal of an antipattern does not guarantee that performance problems are solved, criteria to drive this process towards convergence can be introduced [8].

Table 3 summarizes the performance results obtained by solving the QN models of the different ECS refactored models, after different SRM-TRM pairs have been applied to the original model. The first row of Table 3 indicates the analyzed model. All the following rows refer to refactored models, each identified with ECS followed by the name of the removed antipattern and the number of TRM applied.

The first row of Table 3 reports the evaluation of the QN model for the ECS initial model, as already shown in Table 2. The second row presents the evaluation of ECS as refactored in this section following the EST antipattern solution. Consider, as another example, the ECS∖{Blob−TRM1} row that denotes an ECS model refactored according to the Blob antipattern solution, as specified in TRM1. The second and the third columns of Table 3 show the performance indices of interest, i.e. the response time of the BrowseCatalog and MakePurchase services.

Note that the refactoring actions may add a set of new parameters that need to be evaluated before solving the QN model. For example the EST antipattern is removed by adding new software entity instances for the communication between UserController and Database in the MakePurchase service. The local communication between LocalFacade and Database induces the estimation of two new parameters, that are the demands of CPU and disk. In our experiments we have devised three pairs of values for these new parameters, so to observe the sensitivity of the results to them. For example, the response time of the MakePurchase service varies from 2.32 up to 2.42 seconds in the second row, while the parameters mentioned above vary by about 300%. Hence, in this case the wide variability of parameters does not affect the requirement fulfillment. As opposite, other refactoring actions do not include additional parameters, e.g. the CPS antipattern is solved by re-deploying a

All SRM-TRM pairs are numbered and described in [1].
Figure 6: ECS case study - (Annotated) Software Architectural Model.

Figure 7: ECS case study - (Refactoring-Annotated) Software Architectural Model.
software entity, hence the performance analysis can predict a single value of the MakePurchase response time, that is 2.32 seconds.

Within the ranges of values considered for the additional parameters, Table 3 identifies refactored actions that do not lead actual benefits to both performance indices of interest that are: $ECS \setminus \{Blob - TRM_1\}$, $ECS \setminus \{P\&F - TRM_1\}$ and $ECS \setminus \{P\&F - TRM_2\}$. Hence, the removal of the Pipe and Filter antipattern is not beneficial in any case. Other actions, for some values of additional parameters, are beneficial only to one index, that are: $ECS \setminus \{EST - TRM_1\}$, $ECS \setminus \{Blob - TRM_2\}$ and $ECS \setminus \{CPS - TRM_1\}$. Finally, only one action allows to meet, for certain values of additional parameters, both the requirements, that is $ECS \setminus \{Blob - TRM_2\}$.

Sensible differences among performance results have been experimented across different antipatterns removed, and also across different SRM-TRM pairs for the same antipattern. This consideration supports the usage of our approach that enables analysts to observe such differences and make their decisions on the basis of numerical results.

5. DISCUSSION

The approach presented in this paper highlights the complexity of the identification and removal of performance problems. In this section we focus on some key points that this work has allowed to raise.

Context information. The model elements necessary to detect an antipattern might not be the only ones involved in the model refactoring. Additional information about the context where an antipattern occurs may be necessary to manipulate the model with refactoring actions. Simple examples of this situation could be formulated by expert analysts even without the support of this approach. For example, in order to detect a bottleneck device it could suffice the device utilization, whereas to remove such bottleneck the utilizations of neighbor devices should also be known to equally distribute the workload without introducing adversary effects. This approach allows a systematic definition of such additional information for all performance antipatterns, even in very complex scenarios, and this certainly works towards the automation of the problem solution.

Applicability of refactoring actions. This work has allowed to distinguish among feasible and unfeasible actions. An antipattern definition may refer to design elements (e.g. a software entity sending too many messages) and/or to performance indices (e.g. a hardware device with high utilization). While refactoring actions that refer to design elements can only encounter the obstacles mentioned above, those referring to performance indices often cannot be applied because indices are the results of complex interactions. For example, the split of a software components is a well-defined design refactoring action, whereas the decrease of the throughput of a certain submodel could not be directly mapped to any design refactoring action. Hence, antipatterns could be classified on the amount of performance indices that occur in their definition: the larger this amount is, the harder is the antipattern removal.

Human contribution. The process of removing performance problems can not be fully automated. Such type of process needs to be driven by analysts’ expertise because...
6. RELATED WORK

Differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences [2, 17].

Performance antipatterns deal with the performance issues of software systems. They have been previously documented and discussed in different papers: technology independent performance antipatterns have been defined in [22] and they represent the main references in our work; technology specific antipatterns have been specified in [10, 24].

Enterprise technologies and EJB antipatterns are analyzed in [19]: antipatterns are represented as a set of rules loaded into a detection engine. The monitoring of the software application leads to reconstruct its run-time design and to get system properties. The matching between pre-defined rules and application properties is performed in order to carry out the detection of EJB (i.e. technology specific) antipatterns. However these antipatterns work at the level of code, whereas our approach intends to be model-based and to work at the architectural level.

Very few model-based approaches for software performance diagnosis and improvement have been introduced up today.

In [27] performance problems are identified before the implementation of the software systems, but they are based only on bottlenecks (e.g. the “One-Lane Bridge” antipattern) and long paths. The performance analysis is conducted on Layered Queueing Network (LQN) models. The main limitation of such approach is that it only applies to LQN performance models, hence its portability to other notations is yet to be proven and it may be quite complex.

In [16] 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.

An extensive overview of existing research in the field of software refactoring (not only related to performance problems) is provided in [18]. In literature many approaches often apply the refactoring to the program itself (i.e. the source code), but it is difficult to maintain the consistency between the refactored program code and the other software artifacts. Hence, a need for processes and tools that address refactoring in a more consistent, generic, scalable and flexible way is identified.

In general, there has already been a significant effort in the area of refactoring software design patterns. For example, the specification of UML-based patterns has been addressed in [13] where a pattern specification technique is aimed at defining design patterns as models in terms of UML metamodel concepts. However, as highlighted in this paper, performance antipatterns require appropriate (tailored) techniques to be managed, due to their intrinsic complexity (as combinations of architectural models and performance indices) and their multi-view nature (as combinations of elements from different modeling views). However, as highlighted in this paper, performance antipatterns, as opposite to patterns, require appropriate (tailored) techniques to be managed, due to their intrinsic complexity (as combinations of architectural models and performance indices) and their multi-view nature (as combinations of elements from different modeling views).

7. CONCLUSION

In this paper we have presented an approach, based on role models, that allows to define refactoring actions aimed at removing causes of performance problems in software architectural models. The possibility of considering a set of design alternatives as candidates for the solution of performance problems is a step ahead when compared to the means adopted today that, very often, boil down to the skills and experience of performance analysts. Here we have shown that the solution step can be automated as well, at a certain extent, and a Role-Based Modeling Language is a very promising instrument to address this problem.

Several main directions can be (and are being) followed for the future work. First, we are widening the scope of this approach by introducing SRM-TRM pairs for other known performance antipatterns.

The transformations underlying SRM-TRM pairs are only implicitly defined in this work. Advanced model-driven techniques, such as model differences [4, 21], have been introduced in literature to represent refactorings as difference models. They combine the advantages of declarative difference representations and enable the reconstruction of the final model by means of automated transformations. Hence, they can be applied to our approach to generate the transformations we defined.
Besides, the application of refactoring actions (additions, removals, and modifications) in architectural models must be propagated in a consistent way through different views. A very interesting approach has been introduced in [11] to address consistency among views, and we intend to study how to apply it in the software performance domain.

We have made here the implicit assumption that only one antipattern is considered to be removed at any time. Even more, only one SRM-TRM pair is applied at any time. This could be limiting the whole process, because the combination of pairs (or the combination of antipattern removal) can be a key factor for the success of the process. However, this is a first step in this direction, and we certainly intend to investigate the dependencies among antipatterns in order to define priority rules for the simultaneous solution of antipatterns.

Finally, our approach only considers performance issues in isolation, whereas model refactoring may affect other functional and non-functional properties (e.g. reliability, maintainability, etc.). Consistency and tradeoffs among these properties also deserve to be studied in future.

8. 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 on “Consistent evolution of software artifacts and non-functional models”, and by the VISION ERC project (ERC-240555).

9. REFERENCES