Performance-driven Software Model Refactoring

Davide Arcelli\textsuperscript{a}, Vittorio Cortellessa\textsuperscript{a}, Daniele Di Pompeo\textsuperscript{a}

\textsuperscript{a}University of L’Aquila, L’Aquila, Italy

Abstract

**Context:** Software refactoring is a common practice aimed at addressing requirements or fixing bugs during the software development. While refactoring related to functional requirements has been widely studied in the last few years, non-functional-driven refactoring is still critical, mostly because non-functional characteristics of software are hard to assess and appropriate refactoring actions can be difficult to identify. In the context of performance, which is the focus of this paper, antipatterns represent effective instruments to tackle this issue, because they document common mistakes leading to performance problems as well as their solutions.

**Objective:** In order to effectively reuse the knowledge beyond performance antipatterns, automation is required to detect and remove them. In this paper we introduce a framework that enables, in an unique tool context, the refactoring of software models driven by performance antipattern detection and removal.

**Method:** We have implemented, within the EPSILON platform, detection rules and refactoring actions on UML models for a set of well-known performance antipatterns. By exploiting the EPSILON languages to check properties and apply refactoring on models, we enable three types of refactoring sessions.

**Results:** We experiment our framework on a Botanical Garden Management System to show, on one side, that antipatterns can effectively drive software refactoring towards models that satisfy performance requirements and, on the other side, that the automation introduced by EPSILON-based sessions enables to inspect multiple paths and to propose a variety of solu-

\textit{Email addresses:} davide.arcelli@univaq.it (Davide Arcelli), vittorio.cortellessa@univaq.it (Vittorio Cortellessa), daniele.dipompeo@graduate.univaq.it (Daniele Di Pompeo)
tions.

**Conclusion:** This work demonstrates that automation in performance-driven software model refactoring can be beneficial, and that performance antipatterns can be powerful instruments in the hands of software engineers for detecting (and solving) performance problems usually hidden to traditional bottleneck analysis. This work also opens the road to the integration of well-known techniques for software refactoring driven by functional requirements with novel techniques addressing non-functional requirements like performance.

**Keywords:** Model-Driven Engineering, UML, Software Refactoring, Performance Engineering, Performance Antipatterns,

1. Introduction

Refactoring is nowadays a considerable activity of software development processes, because it can be induced by different reasons such as new requirements, adaptation to multiple contexts, unsatisfactory quality. Software refactoring is a complex task mostly due to the large number of available options and to the lack of effective automated support that can drive towards the preferable ones. In the last two decades, several proposed approaches to software refactoring aimed at improving functional aspects but rarely targeting non-functional ones [1, 2, 3, 4].

On one side, non-functional properties (e.g. performance and energy consumption) are becoming ever more relevant for the success of a software application, especially in the context of distributed platforms that include processing resources by limited capacity like smartphones. On the other side, even when targeting functional aspects, refactoring can unexpectedly affect these properties and lead to violate non-functional requirements.

In this paper we introduce an approach that aims at driving software model refactoring towards the satisfaction of performance requirements. Similarly to other non-functional properties, performance has to be taken into account since the early lifecycle phases [5], because early identification of problems benefits from lower cost solutions. In fact, in the last decade a number of model-driven techniques addressed the identification and solution of performance problems in the early phases of the software life-cycle [6]. Some of these techniques are based on the concept of performance antipattern, which characterizes bad design practices that may jeopardize soft-
ware performance, along with possible refactoring actions aimed to remove them [7].

We have been working on performance antipattern detection and solution under different angles since several years. In [8] we have introduced a logic-based representation of performance antipatterns in order to build non-ambiguous detection on top of it, whereas for the same reason we have introduced in [9] Role Models that represent left and right side of a model transformation that removes a performance antipattern. In [10] we have defined metrics that can drive a process of performance improvement based on antipatterns towards an optimal solution, and in [11] we have tackled the problem of whether it is more advantageous refactoring software models or their corresponding performance models, whereas in [12] we have studied how to combine traditional bottleneck analysis with performance antipattern detection. In [13], and more extensively in [14], we have studied the sensitivity of performance antipatterns to the binding of thresholds appearing in their definition. Finally, in [15] we have extended performance antipattern detection and solution to contexts where a certain degree of uncertainty occur.

This paper roots obviously found into our previous work, as synthesized above, although the basic idea of this paper has only recently appeared in our preliminary work [16, 17], that is to bring performance antipattern detection and solution in an unique environment. In fact, although performance antipatterns have emerged as quite powerful instruments for building automation in the solution of performance problems, the antipattern-based approaches rely on a variety of paradigms, languages, meta-models. This led to fragmentation and lack of interoperability that not only make difficult identifying the most suitable approach for each specific problem, but also limit the potential impact of such approaches. In order to tackle this fragmentation issue we investigated the usage of the EPSILON platform [18], which provides a set of languages, interpreters, and tools, for checking properties and applying refactoring on models.

EPSILON (Extensible Platform of Integrated Languages for model management), developed by University of York, is an open-source platform built on top of the Eclipse Modelling Framework (EMF) [19]. It provides a suite of languages, development tools and interpreters aimed at manipulating models conforming to arbitrary metamodels, by applying well-assessed Model-Driven Engineering techniques (also know as tasks in the EPSILON framework) as model validation, pattern matching, model-to-text transformations, etc.
Among the languages of the EPSILON framework, the Epsilon Object Language (EOL) provides “a reusable set of common model management facilities, atop which task-specific languages can be implemented” [18].

As we have shown in our preliminary work [16, 17], the EPSILON framework should provide an infrastructure suitable for implementing our antipattern-based software model refactoring approach in several ways. In fact, Epsilon Validation Language (EVL), Epsilon Wizard Language (EWL), and Epsilon Pattern Language (EPL), allow to define declarative conditions, namely guard and/or check, and imperative blocks, namely do, that have to be executed if the conditions are satisfied (in case of guard) or not (in case of check). This paradigm matches with the concept of detection and solution of performance antipatterns: the declarative conditions codify antipatterns occurrence rules, whereas in the imperative blocks refactoring actions that might lead to antipatterns solution are explicitly defined. The different execution semantics of the three languages mentioned above allow to provide different automated support to the user, as it will be described in Section 4.

Hence, in this paper we build on top of our preliminary work [16, 17] as follows: we provide more technical details about our framework, we extend the set of performance antipatterns for detection and refactoring, we augment the portfolio of refactoring actions for several antipatterns, we introduce an experimentation on a case study.

In particular, we address here the following research questions:

\textbf{RQ}_1 \textit{Is EPSILON a suitable platform for codifying performance antipattern detection rules and refactoring actions?}

This paper contribution: EPSILON-based code of six well-known performance antipatterns and ten refactoring actions.

\textbf{RQ}_2 \textit{Is it possible to enable different approaches to the software model refactoring within the same environment?}

This paper contribution: Basing on three languages of the EPSILON platform, i.e., EPL, EVL, and EWL, we provide different kinds of refactoring support induced by their distinct execution semantics.

\textbf{RQ}_3 \textit{Does such framework actually lead to improve the performance of a software system?}

This paper contribution: The application of the framework on a UML model case study that demonstrates the effectiveness of our approach in leading towards the satisfaction of performance requirements.
The results of the application of our framework to a Botanical Garden Management System (BGMS) case study provide some evidence of two main aspects of our approach, namely: (i) antipatterns represent sophisticated means to detect causes of performance problems that cannot be detected with traditional bottleneck analysis, in that they combine software aspects scattered across different model views (i.e., static, dynamic, deployment); (ii) automation in a performance-driven refactoring process is undoubtedly necessary to inspect the variety of actions that could potentially remove performance problems, although a fully automated process in some cases could hide aspects that human-driven steps can evidence if opportunely interleaved with automation.

The paper is organized as follows: Section 2 discusses related work. Section 3 provides the background of this work, i.e. performance antipatterns. In Section 4, we first introduce the EPSILON-based framework for performance-driven model refactoring (Section 4.1), i.e., the shaded circle of Figure 2, and then we illustrate the framework capability to translate antipattern detection rules and refactoring actions among the three considered EPSILON languages (Section 4.2). Section 5 describes the application of our framework to an UML model case study that comes from a real world application. Section 6 discusses internal and external threats to the validity of our approach. Finally, Section 7 concludes the paper and mentions future work. In addition, Appendix A describes in detail the considered performance antipatterns.

2. Related Work

Although performance-driven refactoring would be suitable for many application domains, as discussed in the introduction, at the best of our knowledge, this is the first paper that introduces an automated approach for performance-driven software model refactoring. Hence, for a clear separation of concerns, this section is partitioned in two subsections discussing related work on model refactoring and on performance-driven refactoring, respectively.

2.1. Model refactoring

In the last decade, models have been increasingly used since the early phases of software development down to the evolution phase, where software evolves (usually through refactoring steps) for many reasons, like new requirements, changes of context, etc. Hence, software refactoring benefits
from novel approaches not only based on the running code, but also on models that evolve along with code. In fact, the application of refactoring actions on models helps to identify the best refactoring paths before expensively applying them on the code itself.

Most of recent papers in this area deal with model-based refactoring driven by functional properties [1, 2, 3, 4], whereas very few of them consider non-functional properties (e.g., performance and cost) [20]. Non-functional properties are complex to manage because they emerge from the combination of several software aspects that can be static, dynamic, or deployment-related. For this reason, multi-view modeling is required when dealing with these properties.

As reported in a recent systematic literature review (SLR) on UML model refactoring [21], few approaches use multi-views for conducting model refactoring. Furthermore, none of them exploits the UML Deployment Diagram, whereas most of them only work exclusively on Class or Activity Diagrams. We instead base our approach on UML models that include Component, Sequence and Deployment Diagrams together. From the cross-checking of properties in these diagrams, along with performance indices, we are able to detect performance problems in UML models, and to identify refactoring actions that can remove these problems.

Mansoor et al. [22] have presented an approach for multi-view refactoring driven by a multi-objective algorithm (i.e. NSGA-II [23]) towards an optimal set of refactoring actions. They have considered well-known refactoring actions, as previously introduced by Flower et al. [24], in order to improve the design quality of the system (e.g. cohesion and coupling metrics). They apply refactoring actions on a Class Diagram first (i.e. static view), and then they coherently propagate changes to an Activity Diagram (i.e., the dynamic view). As said above, our approach considers, as Mansoor’s one, multi-view UML models, in which we use Component Diagrams instead of Class Diagrams, and Sequence Diagrams instead of Activity Diagrams. In addition, we consider Deployment Diagrams because information about the software deployment is crucial for performance assessment. Finally, we apply refactoring actions coming from performance antipattern detection, because we focus on software performance.

Arendt and Taentzer [25] presented the EMF Refactor tool that is, for some aspects, related to our approach. In fact, both approaches rely on UML models and exploit the Eclipse Modelling Framework (EMF) [26]. EMF Refactor provides a set of model transformation rules, which can be combined
in order to build complex ones. EMF Refactor enables the application of these rules, through graph transformation techniques, to UML and Ecore-based models for refactoring purposes aimed at improving design quality metrics similar to the Mansoor’s approach ones.

2.2. Performance-driven refactoring

Several approaches have been introduced in literature to use performance antipattern knowledge for detecting and removing performance problems in software models [27].

Koziolek et al. [28] have presented an automated approach for performance improvement driven by architectural tactics. The approach describes a multi-objective evolutionary optimization algorithm aimed at searching optimal trade-offs in the design space, in the context of Palladio Component Models (PCMs). The main limitation of this approach is that it is time-consuming because the design space may be huge. Instead, our approach does not look for optimality, but it aims to satisfy performance requirements.

Wert et al. [29] have presented an evolution of Dynamic Spotter [30], one of their previous work. In [29] they have introduced heuristics for measurement-based detection of five well-known performance anti-patterns in inter-component communications. Differently from this approach, our own approach spans over a set of heterogeneous antipatterns (i.e., not only at inter-component communication level), and it is not limited to their detection, but it also proposes refactoring actions that may remove them and may improve performance of the system under analysis.

Xu [31] has presented a prototype named Performance Booster. This approach can be used in the early design phases. Performance antipatterns are detected on a Layered Queuing Network (LQN) obtained from a software model by means of a bi-directional transformation. Refactoring takes place on the performance model, and the corresponding refactored software model is obtained by exploiting transformation bi-directionality. As we have shown in our previous work [11], performance models are more abstract than software models, hence the portfolio of refactoring actions available on the former is much more limited than the one on the latter. In fact, the approach in [31] can only detect and solve bottlenecks, whereas our framework supports a larger and heterogeneous set of antipatterns detection rules and possible solutions.

Parsons et al. [32] analyzed the EJB performance antipatterns. Similarly to our approach, they are represented as a set of rules loaded into a detec-
tion engine, and the application monitoring leads to reconstruct its run-time design and properties. The detection rules are matched on the obtained software model in order to identify the detected EJB antipatterns. Beside the fact that, as opposed to our approach, this approach deals with technologyspecific performance antipatterns, in [32] the application deployment is not considered, so refactoring actions related to re-deployment of components cannot be considered.

Diaz-Pace et al. [33] presented ArchE framework, which assists the software architect during the design to create architectures that meet quality requirements. However, defined rules are limited to improve modifiability only. A simple performance model is used to predict performance metrics for the new system with improved modifiability.

As emerging from this section, our approach differs from those proposed in literature, mainly because it provides antipattern-based refactoring capabilities driven by performance.

3. Background

In this section we provide the main background of this work, represented by Performance Antipatterns.

Since more than a decade, Performance Antipatterns [34, 35] revealed to be strong support for performance-driven software model refactoring, in fact they have been used to “codify” the knowledge and experience of analysts through the representation of potentially bad design patterns that have negative effects on performance of software systems.

Performance antipatterns have been originally defined in natural language [36, 37, 7].

Cortellessa et al. [8] have defined a formal technology-independent interpretation based on first-order logic rules, which identifies a set of system properties under which performance antipatterns occur in a software model. Such rules involve two main ingredients: (i) functions (denoted as $F$) that are used for extracting either design metric values (e.g., number of interface operations) or performance metric values (e.g., device utilization) [13, 14]; (ii) thresholds (denoted as $Th$) that are predefined numerical values which the above metrics are compared to, by means of arithmetical operators, i.e., $<, \leq, =, \geq, >$ [1]. Performance antipattern rules are expressed as first-order

\footnote{Note that also functions can be compared to each other, hence a function may be used}
logic formulae in conjunctive normal form (CNF), i.e. a conjunction of predicates where each predicate is a disjunction of (possibly negated) literals; each literal is a comparison of a metric value (that have to be extracted through its function) and a corresponding threshold.

Code for computing metric values strictly depends on the modelling notation, as extracting a design or performance metric value from a UML [38] model is different from extracting the same metric from a different language, such as a Palladio Component Model (PCM) [39]. For example, the number of operation calls made by a model entity (i.e. a design metric) could correspond to the number of messages sent by a UML Component, whereas it could correspond to the number of ExternalActions invoked by a PCM BasicComponent in the latter case. As a second example, a resource demand (i.e. a performance metric) could be expressed by a specific tag of a MARTE stereotype [40] in the former case, whereas it could be expressed through a built-in PCM model element in the latter case.

For this reason, in order to adopt the antipattern definitions by Cortellessa et al. [8] in our framework, we need to “instantiate” their functions on the targeted modeling notation, which is UML profiled with MARTE. Such an instantiation comes from our experience with performance antipatterns, and it might be used as basis for future instantiations over different modeling notations.

In Table 1 the functions that we have defined and implemented in our framework to extract (design and performance) metrics from software models are listed (2). In particular, the first column reports the signature of the function and the second column provides its description. For example, \( F_{\text{numClientConnects}} \) function takes as input a software component \( c \) and returns an integer that represents the multiplicity of the relationships where \( c \) assumes the role of client.

Given the straightforward relationship between antipatterns’ functions and thresholds, also the latter strictly depend on the modeling notation. This is obvious, since thresholds have to be coherent with the modeling constructs which functions refer to. In other words, also thresholds have to be “instantiated” with respect to the modeling constructs targeted by the

---

2Note that, since in this paper we target UML software models (profiled with MARTE), in the remainder we refer to (a subset of) UML types.
Table 1: Functions specification for model metrics extraction.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ( F_{\text{numClientConnects}}(\text{Component } c) )</td>
<td>It counts the multiplicity of the relationships where ( c ) assumes the client role</td>
</tr>
<tr>
<td>int ( F_{\text{numSupplierConnects}}(\text{Component } c) )</td>
<td>It counts the multiplicity of the relationships where ( c ) assumes the supplier role</td>
</tr>
<tr>
<td>int ( F_{\text{numMsgs}}(\text{Component } c_x, \text{Component } c_y, \text{Service } s) )</td>
<td>It counts the number of messages sent from ( c_x ) to ( swE_y ) in a service ( s )</td>
</tr>
<tr>
<td>float ( F_{\text{maxHuUtil}}(\text{Node } n_x, \text{Type } t) )</td>
<td>It provides the maximum utilization among the hardware devices of a certain type ( t={\text{cpu, io, all}} ) composing the node ( n_x )</td>
</tr>
<tr>
<td>float ( F_{\text{maxNetUtil}}(\text{Node } n_x, \text{Node } n_y) )</td>
<td>It provides the maximum utilization among the network links joining the nodes ( n_x ) and ( n_y )</td>
</tr>
<tr>
<td>float ( F_{\text{maxNetUtil}}(\text{Node } n_x, \text{Component } c) )</td>
<td>It provides the maximum utilization among the network links connecting ( n_x ) overall the nodes with which ( c ) generates traffic</td>
</tr>
<tr>
<td>float ( F_{\text{maxQL}}(\text{Node } n_x) )</td>
<td>It provides the maximum queue length among the hardware devices composing the node ( n_x )</td>
</tr>
<tr>
<td>int ( F_{\text{resDemand}}(\text{Operation } op) )</td>
<td>It provides the resource demand of the operation ( op )</td>
</tr>
<tr>
<td>float ( F_{\text{proxEexec}}(\text{Service } s, \text{Operation } op) )</td>
<td>It provides the probability the operation ( op ) is executed in the service ( s )</td>
</tr>
<tr>
<td>float ( F_{\text{tr}}(\text{Service } s) )</td>
<td>It provides the estimated throughput of the service ( s ) at the steady-state</td>
</tr>
<tr>
<td>float ( F_{\text{rt}}(\text{Service } s) )</td>
<td>It provides the estimated response time of the service ( s ) at the steady-state</td>
</tr>
<tr>
<td>int ( F_{\text{numRemMsgs}}(\text{Component } c, \text{Service } s) )</td>
<td>It counts the number of remote messages sent by ( c ) in a service ( s )</td>
</tr>
<tr>
<td>int ( F_{\text{numRemInst}}(\text{Component } c, \text{Service } s) )</td>
<td>It provides the number of remote instances with which ( c ) communicates in a service ( s )</td>
</tr>
<tr>
<td>int ( F_{\text{numExF}}(\text{Component } c, \text{Service } s) )</td>
<td>It provides the number of exchange formats performed by ( c ) in a service ( s )</td>
</tr>
</tbody>
</table>
corresponding antipatterns functions. Besides, a key point about thresholds concerns their estimation. In fact, as demonstrated by Arcelli et al. [13, 14], threshold values heavily influence the capability of detecting and removing performance antipattern occurrences, especially in contexts where thresholds fuzziness is considered (i.e., where upper and lower bounds on thresholds are provided, instead of single values) [15].

In Table 2 the thresholds on design metrics involved in performance antipattern definitions are listed. In particular, the first column reports the threshold name, the second column provides its description, and the third column provides a possible heuristic for the threshold estimation. For example, the $Th_{\text{maxConnects}}$ threshold represents the maximum number of connections (i.e., Usages or Interface Realizations) for a software component that does not raise an alarm, i.e. the one under which components are assumed not to belong to a performance antipattern. It can be estimated as the average number of connections across the entire set of components in the software system, as from Cortellessa et al. [8]. This means that all components falling under the average are not considered to be critical. The remaining ones could belong to a performance antipattern, depending on the other metrics that concur in the antipattern definition. Note that the heuristics in the third column are the ones that we have adopted in this paper experimentation. However, our approach does not need to be modified if different heuristics are adopted.

Similarly to Table 2, Table 3 reports the thresholds on performance metrics. For example, the $Th_{\text{maxHwUtil}}$ threshold represents the maximum bound for the Nodes utilization, and it can be estimated as the average number of all the utilization values across the entire set of Nodes in the software system plus the corresponding variance, as from Cortellessa et al. [8].

For sake of completeness, Table 4 lists the logic-based representation of the performance antipatterns that we have adopted as the basis of our framework implementation in this paper. Each row represents a specific antipattern that is characterized by two attributes: antipattern name, and its formula, i.e. the first order logics predicate modeling the corresponding antipattern problem.

Basically, such list comes from our previous work where we have introduced logical formulae for all general-purpose performance antipatterns [8]. For sake of simplification, we have excluded here few rare antipatterns plus the ones that need a temporal analysis to be detected. For the same reason, we have refined some antipatterns formulae, as detailed in Appendix A.
Table 2: Thresholds specification for design metrics.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Description</th>
<th>Estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{maxConnects}}$</td>
<td>It represents the maximum bound for the number of connections for a Component</td>
<td>It can be estimated as the average number of Usage relationships, with reference to the entire set of Components in the software system</td>
</tr>
<tr>
<td>$T_{\text{maxMsgs}}$</td>
<td>It represents the maximum bound for the number of messages sent by a Component in a service</td>
<td>It can be estimated as the average number of sent messages, with reference to the entire set of Components in the software system</td>
</tr>
<tr>
<td>$T_{\text{maxRemMsgs}}$</td>
<td>It represents the maximum bound for the number of remote messages in a service</td>
<td>It can be estimated as the average number of remote messages, with reference to the entire set of Components in the software system</td>
</tr>
<tr>
<td>$T_{\text{maxRemInst}}$</td>
<td>It represents the maximum bound for the number of remotely communicating Components in a service</td>
<td>It can be estimated as the average number of remote communicating instances, with reference to the entire set of Components in the software system</td>
</tr>
<tr>
<td>$T_{\text{maxExF}}$</td>
<td>It represents the maximum bound for the number of exchange formats</td>
<td>It can be estimated as the average number of exchanging formats, with reference to the entire set of Components in the software system</td>
</tr>
<tr>
<td>$T_{\text{maxResDemand}}$[i]</td>
<td>It represents the maximum bound for the resource demand of operations</td>
<td>It can be estimated as the average number of resource demands, with reference to the entire set of Operations in the software system, multiplied by a factor greater than 1</td>
</tr>
<tr>
<td>$T_{\text{minResDemand}}$[i]</td>
<td>It represents the minimum bound for the resource demand of operations</td>
<td>It can be estimated as the average number of resource demands, with reference to the entire set of Operations in the software system, multiplied by a factor strictly between zero and 1</td>
</tr>
</tbody>
</table>

For sake of illustration, Figure 1 provides an UML-like graphical example of the Pipe and Filters (PaF) antipattern, which occurs when the slowest filter in a “pipe and filter” architecture causes the system to have unacceptable throughput. In Figure 1a the slowest filter is represented by the Op Operation, and its execution causes a bottleneck in the S Interaction, which shows a throughput lower than a certain threshold. This is due to Op, owned by the C Component that has resource demands (computation, storage, bandwidth) larger than corresponding thresholds. C is manifested by the A Artifact that is deployed on the N Node showing a mean utilization (i.e., $util$) larger than a certain threshold.

A solution to a PaF antipattern occurrence (see Figure 1b) consists of moving the slowest filter to an ad-hoc software component deployed on a specific node. This refactoring is aimed at reducing the utilization of the node where the component owning the largest filter is deployed, and at increasing the throughput of the involved service. In Figure 1b the Op Operation representing the slowest filter has been moved to a new Component $C_{\text{new}}$, which is manifested by a new Artifact $A_{\text{new}}$ deployed on a new Node $N_{\text{new}}$.  

---

12
Table 3: Thresholds specification for performance metrics.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Description</th>
<th>Estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Th_{\text{maxHwUtil}}$</td>
<td>It represents the maximum bound for Nodes utilization</td>
<td>It can be estimated as the average number of all the utilization values, with reference to the entire set of Nodes in the software system, plus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{maxNetUtil}}$</td>
<td>It represents the maximum bound for the Network Link utilization</td>
<td>It can be estimated as the average number of all the bandwidth utilization values, with reference to the entire set of Network Links in the software system, plus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{minNetUtil}}$</td>
<td>It represents the minimum bound for the Network Link utilization</td>
<td>It can be estimated as the average number of all the bandwidth utilization values, with reference to the entire set of Network Links in the software system, minus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{maxQL}}$</td>
<td>It represents the maximum bound for Nodes queue length</td>
<td>It can be estimated as the average number of all queue length values, with reference to the entire set of Nodes in the software system, plus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{maxCpuUtil}}$</td>
<td>It represents the maximum bound for cpu utilization</td>
<td>It can be estimated as the mean value of all the cpu utilization values, with reference to the entire set of Nodes in the software system, plus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{maxIOUtil}}$</td>
<td>It represents the maximum bound for I/O utilization</td>
<td>It can be estimated as the mean value of all the I/O utilization values, with reference to the entire set of Nodes in the software system, plus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{minCpuUtil}}$</td>
<td>It represents the minimum bound for the cpu utilization</td>
<td>It can be estimated as the mean value of all the cpu utilization values, with reference to the entire set of Nodes in the software system, minus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{minIOUtil}}$</td>
<td>It represents the minimum bound for I/O utilization</td>
<td>It can be estimated as the mean value of all the I/O utilization values, with reference to the entire set of Nodes in the software system, minus the corresponding variance</td>
</tr>
<tr>
<td>$Th_{\text{SthReq}}$</td>
<td>It represents the required value for the throughput of a service</td>
<td>It can be estimated as the mean value of throughput overall system services</td>
</tr>
<tr>
<td>$Th_{\text{SrtReq}}$</td>
<td>It represents the required value for the response time of a service</td>
<td>It can be estimated as the mean value of response time overall system services</td>
</tr>
</tbody>
</table>

(a) PaF occurrence.  
(b) A PaF solution.

Figure 1: Pipe and Filter performance antipattern characterization.
Table 4: A logic-based representation of Performance Antipatterns.

<table>
<thead>
<tr>
<th>Performance Antipattern</th>
<th>Logic Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob-Controller</td>
<td>$\exists c_x, c_y \in C, S \in S \mid F_{\text{numClientConnects}}(c_x) \geq Th_{\text{maxConnects}} \wedge F_{\text{numMsgs}}(c_x, c_y, S) \geq Th_{\text{maxMsgs}} \wedge (F_{\text{maxHwUtil}}(P_{xy}, all) \geq Th_{\text{maxHwUtil}} \vee F_{\text{maxNetUtil}}(P_{xy}) \geq Th_{\text{maxNetUtil}})$</td>
</tr>
<tr>
<td>Blob-dataContainer</td>
<td>$\exists c_x, c_y \in C, S \in S \mid F_{\text{numSupplierConnects}}(c_x) \geq Th_{\text{maxConnects}} \wedge F_{\text{numMsgs}}(c_x, c_y, S) \geq Th_{\text{maxMsgs}} \wedge (F_{\text{maxHwUtil}}(P_{xy}, all) \geq Th_{\text{maxHwUtil}} \vee F_{\text{maxNetUtil}}(P_{xy}) \geq Th_{\text{maxNetUtil}})$</td>
</tr>
<tr>
<td>Concurrent Processing Systems (CPS)</td>
<td>$\exists P_x, P_y \in P, OpI \in O, i \in N \mid F_{\text{maxQL}}(P_x) \geq Th_{\text{maxQL}} \wedge (F_{\text{maxHwUtil}}(P_x, cpu) \geq Th_{\text{maxCpuUtil}} \wedge F_{\text{maxHwUtil}}(P_y, cpu) &lt; Th_{\text{minCpuUtil}}) \vee (F_{\text{maxHwUtil}}(P_x, io) \geq Th_{\text{maxDiskUtil}} \wedge F_{\text{maxHwUtil}}(P_y, io) &lt; Th_{\text{minDiskUtil}}) \wedge F_{\text{resDemand}}(OpI)[i] \geq Th_{\text{maxOpResDemand}}[i]$</td>
</tr>
<tr>
<td>Pipe and Filters (PaF)</td>
<td>$\exists OpI \in O, S \in S, i \in N \mid F_{\text{resDemand}}(Op)[i] \geq Th_{\text{resDemand}}[i] \land F_{\text{probExec}}(S, OpI) = 1 \land (F_{\text{maxHwUtil}}(P_c, all) \geq Th_{\text{maxHwUtil}} \vee F_T(S) &lt; Th_{\text{SrThReq}})$</td>
</tr>
<tr>
<td>Extensive Processing (EP)</td>
<td>$\exists OpI_1, OpI_2 \in O, S \in S, i \in N \mid F_{\text{resDemand}}(Op_1)[i] \geq Th_{\text{resDemand}}[i] \wedge F_{\text{resDemand}}(Op_2)[i] &lt; Th_{\text{minOpResDemand}}[i] \wedge F_{\text{probExec}}(S, OpI_1) + F_{\text{probExec}}(S, OpI_2) = 1 \land (F_{\text{maxHwUtil}}(P_c, all) \geq Th_{\text{maxHwUtil}} \vee F_T(S) &gt; Th_{\text{SrThReq}})$</td>
</tr>
<tr>
<td>Empty Semi Trucks (EST)</td>
<td>$\exists c \in C, S \in S \mid F_{\text{numRemMsgs}}(c, S) \geq Th_{\text{maxRemMsgs}} \wedge F_{\text{maxNetUtil}}(P_c) &lt; Th_{\text{minNetUtil}} \wedge F_{\text{numRemInst}}(c, S) \geq Th_{\text{maxRemInst}}$</td>
</tr>
<tr>
<td>Tower of Babel (ToB)</td>
<td>$\exists c \in C, S \in S \mid F_{\text{numExF}}(c, S) \geq Th_{\text{maxExF}} \wedge F_{\text{maxHwUtil}}(P_c, all) \geq Th_{\text{maxHwUtil}}$</td>
</tr>
</tbody>
</table>
4. Performance-driven UML refactoring

Figure 2 illustrates a typical Software Performance Engineering (SPE) process based on UML, in which we have plugged the Performance Antipattern Detection and Model Refactoring Framework that represents the focus of this work.

Usually, a software model does not contain performance attributes and/or indices. However, consolidated techniques exist in literature for: transforming software models into performance models [41], estimating performance attributes like demand vectors [42] and workload [43], and obtaining indices like throughput and response time from performance model solution [44, 45]. This is the goal of the Performance-oriented Model Integration component in Figure 2, namely to get performance indices and set the tagged values of MARTE stereotypes that we assume as their containers in the UML model, thus obtaining a Performance-oriented Software Model. The latter is then given as input to our EPSILON-based Performance Antipattern Detection & Model Refactoring Framework, which includes three engines based on different EPSILON languages (i.e., EVL, EPL, and EWL). Each engine accomplishes the task of: (i) detecting bad design practices that degrade application performance (i.e. performance antipatterns), by verifying codified detection rules, and (ii) removing such bad design by applying codified refactoring actions. Although none of EPSILON languages was conceived for addressing performance-driven model refactoring, in [16] we have identified different
kinds of model refactoring support that they can provide due to their different execution semantics, as summarized in the following.

**Batch refactoring sessions:** Based on EPL, they allow to sequentially execute a set of antipattern detection rules and refactoring actions. The process can be repeated once (i.e., standard mode) or until no more antipattern occurrences are found (i.e, iterative mode).

**User-driven multiple refactoring sessions:** Based on EVL, they allow to execute interactive antipattern detection and refactoring sessions. In fact, after the list of detected antipattern occurrences in the performance-oriented software model is presented to the user, the EVL engine enables a number of available refactoring actions (i.e. fixes) as applicable. Each refactoring is applied to a current temporary version of the software model and, when the user stops the session, the current version is finalized and represents the session output.

**User-driven single refactoring sessions:** Based on EWL, as an element is selected in the modeling environment, antipattern occurrences are immediately detected with respect to the selected element type. Then, the EWIL engine enables the antipattern solutions, among which the user can select the one to apply to the software model, thus producing a refactored model, which represents the session output. A subsequent element selection would trigger a new refactoring session. Due to the strong need of graphical support, these types of sessions are directly integrated in Eclipse-based Graphical Modeling Frameworks, e.g. Papyrus.

In the remainder of this section, we first introduce the EPSILON-based framework for performance-driven model refactoring (Section 4.1), i.e., the shaded circle of Figure 2 and then we illustrate the framework capability to translate antipattern detection rules and refactoring actions among the three considered EPSILON languages (Section 4.2).

### 4.1. Framework Description

The framework architecture is illustrated in Figure 3 which basically shows the main actors that it involves, together with its workflow.

The framework is centered on three performance antipatterns detection and solution engines that provide different interactive support to the designer, and that are respectively based on EPL, EVL, and EWL. The designer selects

---

3https://www.eclipse.org/papyrus/
Figure 3: Architecture of our EPSILON-based framework.

the engine to use in order to perform refactoring sessions starting from an initial software model, i.e. \( M_0 \), which conforms to UML+MARTE metamodel and is given as input to the selected engine. During a refactoring session, a number of new refactored models, i.e., \( M_1, \ldots, M_{n-1} \), can be created until a software model that satisfies performance requirements is obtained, i.e. \( M_n \).

Our framework deals with UML models made of the following diagrams: a Component Diagram that describes the software components and their Interfaces/Operations (\(^\text{4}\)); a Deployment Diagram that describes the allocation of artifacts, corresponding to components, on platform nodes; an Use Case Diagram that describes the actors and the use cases that they can execute; a number of Sequence Diagrams, one for each use case, that describe the system behavior in terms of interactions among components.

Hence, each model \( M_i \) in Figure 3 is required to be an UML model, as specified above, annotated with MARTE stereotypes and tags that represent the information needed to execute the Performance-oriented Model Integration step of Figure 2, that is: (i) the performance parameters required to generate and solve a performance model, and (ii) the performance indices.

\(^\text{4}\)We have recently extended the framework to UML Class Diagrams, but for sake of simplicity we do not introduce this extension here.
required by antipattern detection and refactoring, and that are filled in with the outputs of the performance model solution.

A performance expert has to build the basic knowledge $K_{UML+MARTE}$, which is the UML+MARTE representation of performance antipatterns detection rules and refactoring actions that can be applied to remove them. Such knowledge is then used to produce detection and refactoring code in one of the considered EPSILON languages. Thereafter, the porting engine illustrated in Section 4.2 enables refactoring sessions in the other EPSILON languages.

Figure 4 shows examples of the PaF detection rule as an EPL pattern, an EVL critique condition, and an EWL wizard, respectively. In particular, the PaF occurring condition is verified on UML Operations and it consists of four operations returning boolean values (i.e., $F_{\text{resDemand}}(\text{Th}\_\text{maxResDemand}())$, $F_{\text{probExec}}()$, $F_{\text{T}}(\text{Th}\_\text{SthReq}())$, and $F_{\text{maxHwUtil}}(\text{Th}\_\text{maxHwUtil}()->\text{first}())$) embedded in the PaF formulation [8].

The do block of a pattern/critique/wizard contains a call to the operation titled moveOpNCNN of Table 5, which codifies the PaF refactoring illustrated in Figure 1b [5].

All the EPSILON functions involved in detection rules and refactorings are part of an EOL library that we have developed, i.e., $K_{UML+MARTE}$, which specifies and manipulates objects within the EPSILON platform. The library consists of a set of .eol files, in particular: (i) metric_functions.eol, where functions for extracting model metric values of antipattern detection rules are defined; (ii) thresholds.eol, where functions for extracting threshold values of antipattern detection rules are defined, and (iii) a .eol file for each UML type involved in software models (e.g., component.eol), which contains refactorings and supporting operations for that type (e.g., moveOpNCNN() refactoring and getNearLessUsedNodes()). This introduces a high degree of modularity which eases the reuse and the isolation of EPSILON code writing issues.

Each detection rule has been designed as a logic formula in conjunctive normal form of predicates implemented as calls to boolean EOL functions, whose code excerpts are reported in Listing 1.

---

5The EPSILON code is available in the Git repository at [https://git.io/vy3ZY](https://git.io/vy3ZY).
(a) Excerpt of an EPL pattern for PaF.

(b) Refactoring session with EPL.

(c) Excerpt of an EVL critique for PaF.

(d) Refactoring session with EVL.

(e) Excerpt of an EWL wizard for PaF.

(f) Refactoring session with EWL.

Figure 4: Refactoring engines of EPL, EVL, and EWL.
The `F_resDemand` function verifies if there exists a pair of UML Operations (line 6), i.e. `opGreater` and `opLesser` (lines 4-5) such that: (i) `opGreater` is owned by the UML contextual component, whereas `opLesser` is owned by the contextual component itself or a different component; (ii) the resource demand of `opGreater` is greater or equal to the `th_maxResDemand` input threshold, whereas the resource demand of `opLesser` is below `th_minResDemand`. An example of function for extracting input threshold values is reported in Listing 2, with respect to `th_maxResDemand`, which basically sets
the value of the latter to 20.0 (line 3) and returns it (line 4). Note that the `th_maxOpResDemand` threshold is a list of Reals; this is because, in general, a resource demand may span over several hw/sw resources (e.g., computation, storage, cpu units). For sake of simplicity we assume it is one value representing the “amount of service units” requested by an Operation to the hardware node where its owning component is deployed to.

Listing 2: `th_maxOpResDemand` input threshold function into the EOL library.

```java
operation th_maxOpResDemand() : List<Real> {
  var th_maxResDemand : List<Real> = new List<Real>;
  th_maxResDemand.add(20);
  return th_maxResDemand;
}
```

The `F_probExec` function verifies if there exists a UML alternative fragment with two interaction operands such that: (i) they call `opGreater` and `opLesser`, respectively (lines 13-14); (ii) their execution probability (expressed by the `prob` tag of «GaStep») sum to 1 (line 15).

The `F_maxHwUtil` function verifies if there exists an instance of the contextual component deployed to a UML node showing an utilization (expressed by the `utilization` tag of «GaExecHost») greater or equal to the `th_maxHwUtil` input threshold (line 22).

The `F_RT` function verifies if there exists a UML use case (line 33) that: (i) contains the alternative fragments with total probability equal to 1 (see the `operationMap` of line 29); (ii) shows a response time (expressed by the `respT` tag of «GaScenario») greater or equal to the `th_SrtReq` input threshold (line 30).

In order to enable a performance-oriented model refactoring based on antipattern detection, a portfolio of refactoring actions must be available. The latter ones should have different nature, different complexity, and should apply to different element types, in order to have effective refactoring options.

Table 5 gives a complete view of the refactoring portfolio that we have built in our framework. For each refactoring: the first column reports the contextual UML element type that the refactoring can be applied to; the second column contains the refactoring name; the third column provides a description of the refactoring; the fourth column reports the performance antipattern that can be targeted by the refactoring, accordingly with its reference element (i.e., the corresponding contextual UML element type).

Going through Table 5, the first five refactoring actions target one or more performance antipatterns, as they have been conceived to be “antipattern-
<table>
<thead>
<tr>
<th>Refactoring Context (UML Element)</th>
<th>Refactoring Name</th>
<th>Description</th>
<th>Performance Antipattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>moveOpNCNN</td>
<td>Moves the most demanding Operation of the contextual Component to a new Component which is deployed to a new Node.</td>
<td>Blob</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td>Moves the contextual Operation to a new Component which is deployed to a new Node.</td>
<td>FaP, E3</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>Moves the most demanding Operation of the most demanding Component deployed to the contextual Node to a new Component which is deployed to a new Node.</td>
<td>CPS</td>
</tr>
<tr>
<td>Component</td>
<td>moveOpNCLN</td>
<td>Moves the most demanding Operation of the contextual Component to a new Component which is deployed to the less used Node.</td>
<td>Blob</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td>Moves the contextual Operation to a new Component which is deployed to the less used Node.</td>
<td>PaF, EP</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>Moves the most demanding Operation of the most demanding Component deployed to the contextual Node to a new Component which is deployed to the less used Node.</td>
<td>CPS</td>
</tr>
<tr>
<td>Component</td>
<td>moveOpLC</td>
<td>Moves the most demanding Operation of the contextual Component to the less demanding Component.</td>
<td>Blob</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td>Moves the contextual Operation to the less demanding Component.</td>
<td>FaP, E3</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>Moves the most demanding Operation of the most demanding Component deployed to the contextual Node to a new Component which is deployed to the less demanding Component.</td>
<td>CPS</td>
</tr>
<tr>
<td>Component</td>
<td>moveCompNN</td>
<td>Moves the Component with the most demanding Operation to a new Node.</td>
<td>Blob</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td>Moves the Component with the most demanding Operation to a new Node.</td>
<td>FaP, E3</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>Moves the Component with the most demanding Operation to a new Node.</td>
<td>CPS</td>
</tr>
<tr>
<td>Component</td>
<td>moveCompLN</td>
<td>Moves the Component with the contextual Operation to the less used Node.</td>
<td>Blob</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td>Moves the Component with the contextual Operation to the less used Node.</td>
<td>FaP, E3</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>Moves the Component with the most demanding Operation among the ones hosted by the contextual Node to the less used Node.</td>
<td>CPS</td>
</tr>
<tr>
<td>Component</td>
<td>reduceUsages</td>
<td>Reduces the Usage relationships of the contextual Component by moving a number of used Operations to the contextual Component.</td>
<td>Blob</td>
</tr>
<tr>
<td>Operation</td>
<td>reduceInterfaces</td>
<td>Reduces the Interfaces of the contextual Component by moving a number of Operations to the Components that use them.</td>
<td>Blob</td>
</tr>
<tr>
<td>Component</td>
<td>decomposition</td>
<td>Replaces the contextual Operation with a new version, which is less demanding than the former version and which may be executed in times in place of the latter.</td>
<td>FaP</td>
</tr>
<tr>
<td>Operation</td>
<td>applyFacade</td>
<td>Applies the Facade pattern [] with the main goal of reducing the number of messages sent over the network, by introducing facades between the contextual Component and the remote Components involved in a certain Interaction. A façade accumulates messages from the contextual Component and forwards a single message to the other façades, which, in turn, deliver the original messages to the corresponding recipient remote Components.</td>
<td>EST</td>
</tr>
<tr>
<td>Component</td>
<td>applyStdFormat</td>
<td>Unifies the data formats adopted by the contextual Component, in a certain Interaction.</td>
<td>FaP</td>
</tr>
</tbody>
</table>

Table 5: Portfolio of refactoring actions.
independent” in the sense that they do not act on a single specific aspect of a certain antipattern, differently from the last five refactoring actions which are “antipattern-specific”.

Given such distinction, it comes straightforward why the former refactoring can be applied to different UML types, as specified in the first column, similarly to the Java method overriding mechanism. Conversely, due to the fact that the latter five refactoring actions operate on antipattern-specific aspects, each of them is applicable only to one performance antipattern and, consequently, only to its corresponding UML element specified in the first column.

For example, different versions of moveOpNCNN exist, and they basically move a certain Operation to a new Component which is deployed to a new Node. Such refactoring has a different effect if applied to different UML types, in particular:

- If moveOpNCNN is called from the context of a Component, then the moved Operation is the most demanding among the ones of the contextual Component \(^6\). This version of moveOpNCNN targets the Blob antipattern, by calling it from the Blob context, i.e., Component.

- If moveOpNCNN is called from the context of an Operation, then the moved Operation is the contextual one. This version of moveOpNCNN targets the PaF and EP antipatterns, by calling them from their corresponding context, which is Operation for both of them.

- If moveOpNCNN is called from the context of a Node, then the moved Operation is the most demanding one of the most demanding Component deployed to the contextual Node. This version of moveOpNCNN targets the CPS antipattern, by calling it from the CPS context, i.e., Node.

Differently from moveOpNCNN, reduceUsages is an “antipattern-specific” refactoring. In particular, it moves to the contextual Component a number of used Operations. Such a refactoring is specific for the Blob antipattern, hence only one version exists, and it works in the context of the Blob, i.e., Component.

\(^6\)Given a set of Operations, the most demanding Operation is defined as the one having the highest average demand.
Refactoring actions moveOp and moveComp are mainly aimed at reducing as much as possible the load of a Node whose utilization is unacceptable, as the latter represent a key-metric in any performance antipattern definition. For example, by applying a moveComp refactoring to the overloaded Node of a CPS occurrence, the load of the latter shall decrease, thus possibly bringing to performance improvement. However, such a load distribution might also bring to performance degradation if, for example, the load distribution causes the saturation of a destination Node.

Refactoring actions do not necessarily target performance metrics. For example, reduceInterfaces and reduceUsages for the Blob move a number of Operations from and to the contextual Component (i.e., the Blob) in order to overcome corresponding design thresholds. However, as for moveOp and moveComp, moving such Operations induces the side effect of modifying the utilization of the involved Nodes.

We show the Epsilon code for EVL, EPL, and EWL, with respect to the Extensive Processing performance antipattern (EP) [8]. For sake of illustration, we do not show more code. However, in order to address RQ1, we have implemented: (i) detection rules of the six performance antipatterns of Table 4, (ii) the five “antipattern-independent” refactoring actions of Table 5 (see fixes in Listing 3), (iii) the remaining five “antipattern-specific” refactoring actions.

Listing 3 reports the EVL code for EP, i.e. the detection rule and the five “antipattern-independent” refactorings, whereas Listing 4 (resp. Listing 5) reports the pattern (resp. wizard) that correspond to the first EVL fix (line 6).

Going deep in refactoring actions implementation, let us consider the fix named moveOpNCNN of Listing 3, and the corresponding pattern/wizard in Listings 4 and 5, respectively. This refactoring consists in invoking a specific function, namely moveOpNCNN, on the component owning the most demanding operation (see lines 2-3 of Listing 6). Such function invokes a homonym one (shown in Listing 7) on the most demanding operation that, through the invocation of other supporting functions, is moved to a new component (lines 3-4), which is deployed to a new node (line 5). The dynamic view is finally modified accordingly to the previous changes (line 6). The refactoring ensures that the behaviour of the system is still guaranteed. First of all, a new UML Lifeline is introduced, representing the new Component (namely, splitted-Component); then, the UML Message of the most demanding operation is retrieved, and its recipient lifeline is modified with the new one. Afterwards,
for each element between such message and the first corresponding reply message, all recipients and senders (i.e. those lifelines that receive a message or send a message in the considered interval) are modified according to the refactoring. Such refactoring aims at lightening the load on the node where the component owning the most demanding operation is deployed to, and at reducing the response time of the service(s) that involve the most demanding operation.

Listing 3: Extensive Processing in EVL.

```java
context Operation {
    critique ExtensiveProcessingPA {
        check:
            not (self.F_resDemand(th_maxResDemand(), th_minResDemand()) and self.F_probExec() and (self.F_maxHwUtil(th_maxHwUtil()) or self.F_RT(th_SrtReq())))
        message: "EP <" + self.type.name + "+ " + self.name
        fix {
            title: "Move it to a new Component deployed to a new Node"
            do {
                self.moveOpNCNN();
            }
        }
        fix {
            title: "Move it to a new Component deployed to the less used neighbour Node"
            do {
                self.moveOpNCLN();
            }
        }
        fix {
            title: "Redeploy its owning Component " + self.name + " to the less used neighbour Node"
            do {
                self.moveCompLN();
            }
        }
        fix {
            title: "Change its owning Component from " + self.name + " to the one with the lowest demand"
            do {
                self.moveOpLC();
            }
        }
        fix {
            title: "Redeploy its owning Component to a new Node"
            do {
                self.moveCompNN();
            }
        }
    }
}
```
4.2. Translations among EPSILON languages

As mentioned before, the Porting Engine of Figure 2 implements the automated translation of the code for performance antipatterns detection rules and refactoring actions among EPL, EVL, and EWL. It consists of a set of

Listing 4: Extensive Processing in EPL.

```plaintext
pattern ExtensiveProcessingPA
mainRole : Operation {
  match :
    mainRole . F_resDemand ( th_maxResDemand () , th_minResDemand () ) and
    mainRole . F_probExec () and ( mainRole . F_maxHwUtil ( th_maxHwUtil () ) or
    mainRole . F_RT ( th_SrtReq () )
  onmatch {
    mainRole . moveOpNCNN ();
  }
}
...
```

Listing 5: Extensive Processing in EWL.

```plaintext
wizard ExtensiveProcessingPA {
  guard :
    self . isTypeOf ( Operation ) and
    self . F_resDemand ( th_maxResDemand () , th_minResDemand () ) and self .
    F_probExec () and ( self . F_maxHwUtil ( th_maxHwUtil () ) or self . F_RT (
    th_SrtReq () )
  title : "Move it to a new Component deployed to a new Node"
  do {
    self . moveOpNCNN ();
  }
}
...
```

Listing 6: moveCompNN refactoring for UML Components.

```plaintext
operation Component moveCompNN () : Boolean {
  var criticalOp = self . getCriticalOperation ();
  criticalOp . moveOpNCNN ();
  return true;
}
```

Listing 7: moveOpNCNN refactoring for UML Operations.

```plaintext
operation Operation moveOpNCNN () : Boolean {
  var sourceComponent : Component = self . getOwner ();
  var splittedComponent = self . splitComponent ();
  splittedComponent . moveOperation ( self );
  splittedComponent . deployOnNewNodes ( sourceComponent );
  self . dynamicFixing ( sourceComponent );
  return true;
}
```
mappings among EVL/EWL/EPL abstract syntaxes. Such an automated translation is very important for our framework, because it reduces the effort in EPSILON code writing. In fact, it allows the performance expert to manually write code only for one of the three sub-engines (e.g., EPL), and then to generate the code for the other two, if she wants to exploit the different kinds of refactoring support that their execution semantics provide. Note that, without such an automated code porting, the performance expert has to manually write code for each considered EPSILON language, which means implementing the same antipatterns detection rules and refactoring actions three times, as the mapping among languages is not straightforward. For example, each EPL pattern contains only one imperative block (i.e., onmatch) for specifying the refactoring actions to apply when the corresponding antipattern detection rule is matched on the model, whereas each EVL rule can contain one or more imperative blocks (i.e., fix) that codify several refactoring actions.

The porting feature of our framework contributes to answer RQ$_2$, as it demonstrates that it is possible to translate EPSILON among the three considered engines, i.e., among their different kinds of refactoring support.

4.2.1. Mapping Definition

For sake of illustration, Figure 5 shows mappings among EPL (at the left-hand side), EVL (at the center), and EWL (at the right-hand side), in terms of their concrete syntaxes (7). Mappable elements are represented by boxes, and mappings are expressed with lines between boxes. When a Performance Expert writes the code in a certain EPSILON language, she must conform to the structure reported in Figure 5 in order to enable a correct automated translation of her code. Note that not all the constructs of an EPSILON language are mappable to any other language. In other words, there is an information loss with respect to some constructs. For example, EVL message cannot be mapped to any other construct of EPL and EWL.

In the remaining of this section we describe the mapping in detail.

Each critique into a context maps to one or more wizards and to one or more pattern (and vice-versa), as follows: each fix of a critique maps onto a homonym pattern having: (i) a role named mainRole whose type is

7Note that, with respect to our purposes, we target part of the syntaxes of the three EPSILON languages, e.g., we ignore annotations.
the same of the context owning the critique; (ii) the same detection rule of the critique; (iii) the same refactoring of that fix. The same arguments can be applied to critique and wizard, with a difference: the condition self.isTypeOf(<type>) is added to the guard of the wizard rather than adding mainRole as for EPL. Moreover, the critique and the wizard have the same title.

A wizard maps 1:1 to a pattern (and vice-versa). Note that the latter contains a title which cannot be specified in a pattern. However, this does not represent a problem, because patterns are executed in batch, hence no output is needed to be provided to the end-user.

The do blocks inside fixes, wizards, and the onmatch blocks inside patterns, map one to the other.

Concerning EVL it is worth to notice that:

- We use critiques instead of constraints. This is because, when executed, a critique manifests as a warning in the EVL console, whereas a constraint manifests as an error. We think that the concept of performance antipattern occurrence is closer to the concept of warning rather than error, due to the fact that in general the presence of an antipattern does not necessarily compromise performance requirements fulfillment.

- The check of a critique is negated with a not operator. This is because in EVL the warning of a critique is reported whenever the corresponding check is not verified (i.e., the check has been conceived by EPSILON developers as a condition that must hold in the model), hence such negation is needed for reporting that the detection rule of an antipattern has been verified on the software model.
4.2.2. Mapping Implementation

The mappings among the three considered EPSILON languages have been implemented by a Java program that works on the Abstract Syntax Trees (AST) of such languages. This Java program produces syntactically correct files that can be processed by our framework. In particular, the Java program is equipped with six methods EXL2EYL, with $X \in \{P, V, W\}, Y \in \{P, V, W\},$ and $Y \neq X$ (e.g., from EVL to EWL, see Listing 8) that span over a set of Java classes. According to the predefined mappings, they manipulate the AST instantiated on the .exl input file and produce another AST that conforms to the syntax of the .eyl output file. The obtained AST is then transformed to text by the ast2file method of the PortingUtil class (see Listing 9), by exploiting a lightweight extension of the EPSILON platform that we have performed. In particular, an Interface named ASTRewrite owning a method named rewrite() has been introduced, and a dependency has been added from the EPSILON AST class (i.e. the EPSILON class for ASTs) and ASTRewrite, where the former implements the latter. Moreover, a default implementation of the rewrite() method has been introduced into the EPSILON AST class. Each EPSILON domain class that extends AST overrides the rewrite() method, thus returning the syntactically correct string for that class. Due to this extension, the text for the output file is correctly obtained by construction starting from the root of an AST and going deeply throughout its children.

Listing 8: Excerpt of the Evl2Ep1 class.

```java
package it.spe.disim.univaq.porting.evl2exl;
...
public class Evl2Ep1 extends Exl2Ey1 {
...
  public static AST evl2ep1(AST evlAST) {
    AST eplAST = PortingUtil.createModuleAST(...);
    for(AST context : AstUtil.getChildren(evlAST, EvlParser.CONTEXT)) {
      for(AST critique : AstUtil.getChildren(context, EvlParser.CRITIQUE)) {
        for(AST fix : AstUtil.getChildren(critique, EvlParser.FIX)) {
          Pattern pattern = PortingUtil.createPattern(critique.getText());
          Role role = PortingUtil.createRole(EplParser.ROLE, "");
          NameExpression mainRole = PortingUtil.createNameExpression("mainRole");
          TypeExpression typeExp = (TypeExpression) context.getFirstChild();
          role.setFirstChild(mainRole);
          role.addChild(typeExp);
          pattern.setFirstChild(role);
        }
      }
    }
  }
  ...
}
```

The EPSILON code is available in the Git repository at [https://git.io/vy3ZY](https://git.io/vy3ZY).
Listing 9: Excerpt of the PortingUtil class.

```java
package it.spe.disim.univaq.porting.util;
...
public class PortingUtil {
  ...
  public static String ASTRewrite(AST ast) {
    String toString = "";
    for (AST child : ast.getChildren()) {
      toString += child.rewrite();
    }
    return toString;
  }
  ...

  public static void ast2file(AST ast, String filename, String extension) {
    try {
      FileUtils.writeStringToFile(new File(basePath+extension+"/"+filename+extension), ASTRewrite(ast));
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

5. Experimentation

In this section we describe the application of our framework to an UML model case study that comes from a real world application.

This section is structured as follows: Section 5.1 illustrates the case study, i.e., its application domain, the UML and MARTE model, and the corresponding Queuing Network that we have generated to obtain performance indices. Section 5.2 describes the performance antipatterns detection phase on the initial model. Section 5.3 describes the model refactoring process by showing the application of refactoring actions over multiple steps, as executed within the three different sessions provided by our framework. Finally, Section 5.4 discusses the implications of our experimental results with respect to RQ$_2$ and RQ$_3$. 

30
5.1. Case Study Description

The application domain that we have considered is a Botanical Garden, which is a garden dedicated to the collection, cultivation and display of a wide range of plants labelled with their botanical names. It may contain specialist plant collections such as herb gardens, plants from particular parts of the world, and so on. There may be greenhouses, shade-houses, again with special collections such as tropical plants, alpine plants, or other exotic plants. Visitor services at a botanical garden might include tours, educational displays, art exhibitions, book rooms, open-air theatrical and musical performances, and other entertainment.

Figures 6 through 10 depict the UML+MARTE diagrams, as required from our framework (see Section 4.1), that represent the reference model of a Botanical Garden Management System (BGMS). The Component Diagram is reported in Figure 6, which represents the static view of the BGMS. It is worth to notice that in this view we represent each Operation of a Component using an UML Interface element, so that the designer can immediately identify the connections among Components. We have stereotyped each Operation as GaStep to enable the annotation of performance characteristics on it. For example, using GaStep tags (i.e., servCount and servDemand, the type and amount of hardware service that is required by the Operation), we can discover the most demanding Operation in the detection phase. Several Components and Operations are also stereotyped as PaRunTInstance, which is used as a detection filtering. In fact, all UML elements stereotyped in this way will not be considered from the detection engine as possible origins of performance antipatterns. For example, in this specific case Application is stereotyped as PaRunTInstance because it runs on the client side of BGMS, which is considered out of designer control.

In Figure 7 we have reported the Deployment Diagram of the BGMS. In this view we use, for performance purposes, RtUnit and GaExecHost stereotypes for annotating the queueSize and the utilization values as carried out by performance analysis. Furthermore, in this view we use other MARTE stereotypes, which are not presented in this case study, like GaCommChannel that

\(^9\)As UML model, we have selected the best project among the ones assigned this academic year to our master students, in the context of the Advanced Software Engineering course at University of L’Aquila.

\(^{10}\)Some diagrams also include stereotypes from a specific UML profile related to botanical garden domain that students have introduced (e.g. «Sensor» in Figure 6).
is applied over the Communication Path (i.e., a link between two Nodes) to indicate the usage of network through its utilization tag, and GaCommHost for representing a possible network host (e.g., a network router). Similarly to Components and Operations in the Component Diagram, here Nodes can be stereotyped as PaRunTInstance in order to exclude them from the detection step. For example, in this case Mobile will be treated in this way because it represents the client side of BGMS.

Figures 8, 9, and 10 represent the dynamic view of the BGMS. Figure 8 depicts the Use Case Diagram, where each Use Case represent a software service. In this diagram, we use the GaScenario stereotype for annotating the response times and throughputs of considered services, by means of respT and throughput tags, respectively, as carried out by performance analysis. Moreover, we introduce one Interaction for each considered Use Case, that are the Sequence Diagrams depicted in Figures 9 and 10. The former Sequence Diagram describes the Visitor access to the Greenhouse, whereas the latter one illustrates the setting changes inside the botanical garden (e.g., when watering is required).

Summarizing, GaStep, GaExecHost, RtUnit, GaCommHost, GaCommChannel, and GaScenario, are needed to annotate the model with performance input–output information (e.g., Operations service demands, Node service times, network latencies, Node queue lengths, services response times). In addition to them, the PaRunTInstance stereotype can be used to annotate Components/Operations/Nodes intended to be excluded as candidate origins of performance antipatterns.

Finally, another stereotype appears in the UML diagrams, that is Allocated. It has a similar role of PaRunTInstance for sake of refactoring, because UML elements annotated as Allocated are not considered from the refactoring engine.

The combination of PaRunTInstance and Allocated in the hands of designers allow to selectively decide the model areas where detection and refactoring engines have to operate. We need two stereotypes because there could be an element that the designer likes to know whether it is an antipattern origin, but on which she/he cannot do any refactoring action (e.g., because it is a legacy component).

In Figure 11 the Queueing Network (QN) model of BGMS is reported, as obtained from transforming the UML model. We have used Java Mod-
Figure 6: BGMS - Component Diagram

Figure 7: BGMS - Deployment Diagram
elling Tool (JMT) \(^{11}\) for QN simulation purposes. Two classes of jobs circulate in the model: (i) Visitors, modeled as an open class, execute the AccessGreenhouse service, whereas (ii) Administrators, modeled as a closed class, execute the ChangeSettings service.

Four different types of nodes occur in the QN model: (i) a Source node, namely Visitors, in which the AccessGreenhouse jobs are produced; (ii) a Sink Node, namely VisitorsEnd, from which these jobs leave the system; (iii) a Delay node, namely Administrators, which manages ChangeSettings jobs; (iv) Service nodes, i.e. all the remaining ones, which represent the platform components that we consider. In particular, platform components can be mapped back onto the Deployment Diagram as follows: CPU-G is the CPU of the Gateway Node; CPU-S, Disk-S1 are the CPU and Disk of Server Node, respectively; CPU-C is the CPU of the Camera Node; CPU-M and Disk-M are CPU and Disk of Mobile Node; CPU-DB and Disk-DB are CPU and Disk of Database node, respectively.

\(^{11}\)http://jmt.sourceforge.net/
Figure 9: BGMS - AccessGreenhouse Sequence Diagram
Figure 10: BGMS - ChangeSettings Sequence Diagram
5.2. Performance Antipatterns Detection

Table 6 reports the performance antipattern detected on the initial BGMS model. The first column reports antipattern acronyms and the corresponding UML element type between brackets, whereas the second column reports the BGMS model element instance(s) generating the antipattern.

The BGMS contains two Blob-controller occurrences (i.e., Components): GreenhouseManagement and Guide. In the remaining of the paper we refer to them as Blob(s), since they are the unique Blob case occurring in the BGMS.

Three CPS also occur, all refer to their corresponding overloaded UML Node, i.e., Server. The underloaded Nodes of such CPS occurrences are Camera, Database, and Mobile, respectively.

The Operation named checkCredentials, which is owned by Security, originates a PaF occurrence, mainly due to its high demand.

EP, EST, and ToB antipatterns do not occur in the BGMS; it is worth to notice that an EST occurrence would have been detected as originated by Application, but we have stereotyped the latter as PaRunTInstance because Application is deployed at the client-side that we consider out of our control.

Summarizing, the initial BGMS model shows the following performance antipattern occurrences: two Blobs, one (triple) CPS, and one PaF.

A preliminary manual detection of actual performance antipattern occurrences in the BGMS has been conducted, and it plays the role of oracle for
validating the automated antipattern detection phase of our framework. The
detected performance antipattern occurrences in the BGMS exactly matched
the actual ones. This means that the former represent true positives and that
the approach did not miss any antipattern occurrence in this case (i.e., no
false negatives have been detected) \(^{12}\). Although in our case study we can
rely on 100% recall (i.e., no false negatives – all the actual antipattern occur-
rences have been detected) and 100% precision (i.e., no false positives – all
the detected antipattern occurrences are actual ones), obviously this result
cannot be expected in other cases.

However, we would like to remark that in a stochastic context, like the
performance engineering one where our framework operates, precision and
recall are less relevant than they are in a functional design context. In fact,
the definitions of performance antipatterns, as it has been shown before in
this paper, often involves performance metrics and thresholds that depend on
stochastic aspects of the system (e.g., the user profile), whereas the definitions
of functional antipatterns, such as Spaghetti Code \([46]\), only involve static
aspects of software (e.g. lines of code) that can be directly mined from the
code. Hence, values assumed by precision and recall in a performance context
only give a rough idea of detection capabilities. Such capabilities have in
fact to be validated by the analysis of refactored models where performance
antipatterns have been removed.

Nevertheless, it is appropriate to ask whether it is worthwhile to pursue
precision and/or recall in our context.

Usually, precision and recall do not grow together \([14]\), thus it should be
pre-defined whether high recall or high precision is preferred. Intuitively, the

\(^{12}\)We remind that recall \((r)\) and precision \((p)\) have been introduced to analyze the
amount of false positives and negatives, and they are defined as follows:
\(r = \frac{TP}{TP + FN}\),
\(p = \frac{TP}{TP + FP}\), where TP, FN and FP are the number of true positives, false negatives
and false positives, respectively.
extreme case where an antipattern detector raises alarms on all model elements results in a 100% recall, but the corresponding precision is minimum. Conversely, in case it never raises alarms the recall is minimum whereas it results in a precision of 100% (which is distorted, because no antipattern has been detected). This behavior heavily depends on the threshold settings. Hence, the tighter the thresholds are the less false positives and the more false negatives are detected (i.e., precision increases and recall decreases), and vice versa.

For sake of refactoring, targeting false positive system entities that are not actual origins of performance antipatterns (i.e., in case of high recall) represents a waste of time that, in the best cases, results in negligible improvement of performance and, in worst cases, may lead to performance degradation.

As opposite, cutting out from the refactoring phase false negative system entities that are actual origins of performance antipatterns (i.e., in case of high precision) may only bring a slowdown of the performance engineering process towards requirements fulfillment because some promising candidates are not considered. However, this case is unlikely to lead performance degradation because the refactoring targets are very likely true positives.

On this basis, for performance threshold estimation, we have preferred restrictive heuristics that originate tighter thresholds. For example, the estimation of $T_h_{maxH\text{wUtil}}$ in Table 3 adds the variance to the average utilization overall Nodes, so making the threshold tighter than the simple average.

Finally, techniques can be introduced to improve the precision of performance antipattern detection, such as the one based on guilt that Cortellessa et al. have introduced in [10].

5.3. Refactoring

In this section we apply several refactoring actions from our portfolio (Table 5) to the reference UML model, showing how our framework allows to explore the solution space.

Table 7 reports an exhaustive overview of the refactoring actions that are available after the performance antipatterns detection on the initial BGMS.

At this point, it is worth to remark the relevance of the $Allocated$ stereotype. For example, it prevents Camera from being considered as the less demanding Component in the refactoring step, which would have instead been the target of moveOpLC refactoring action. Similar arguments apply to moveOpNCLN and moveCompLN refactoring actions if the $Allocated$
stereotype would have not been applied to Camera Node. In this case, the latter would have been the destination of a move refactoring action.

Figure 12 shows a refactoring tree generated by the experimentation that we have performed on our case study. In order to address \( RQ_3 \), we have worked on different refactoring actions with the aim of meeting two predefined performance requirements, i.e., **Performance Requirement 1** and **Performance Requirement 2**, stating that the average response times of AccessGreenhouse (AG) and ChangeSettings (CS) scenarios must not exceed 30 and 40 seconds, respectively.

Since an exhaustive exploration of the solution space would not be feasible, for sake of illustration the tree embraces a subset of the refactoring actions of Table 7. On one side, it embraces the ones spanning over several antipattern occurrences, i.e. the highlighted ones. These include the “Blob-specific” action reduceUsages and several “antipattern-independent” ones that move Operations or Components to a new Node, as shown by the numbers of tree nodes reported in Table 7. The highlighted refactoring actions are applied to the initial BGMS (i.e., the root of the tree) and result into the first-level nodes. On the other side, the non-highlighted actions are applied to refactored models that correspond to second- and third-level nodes of the tree, with the goal of possibly targeting different model elements than the previous ones.

The rationale behind this choice of refactoring actions is that we would like to show the results coming from different kind of refactoring actions (“antipattern-specific” vs “antipattern-independent” ones) and from different refactoring actions of the same kind. However, other criteria could drive the exploration of the solution space, both in terms of antipattern occurrences prioritization as in Trubiani et al. [10] and in terms of antipattern occurring probability and refactoring action effectiveness as in Arcelli et al. [15].

### Nodes of the refactoring tree.

The tree has its root in the BGMS, and each other node represents a refactoring step. Each node is annotated with the response times resulting from the application of the refactoring actions in the path from the root to that node. Node response times of Figure 12 are preceded by a √, if they meet the two predefined performance requirements, and by a ✗, otherwise.

The refactoring tree is composed by 12 nodes and, apart from the root that does not satisfy the requirements,
Table 7: Available refactoring actions for the initial BGMS.

<table>
<thead>
<tr>
<th>Performance Antipattern</th>
<th>BGMS model element</th>
<th>Context</th>
<th>Available refactoring actions</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob</td>
<td>Greenhouse Management</td>
<td>Component</td>
<td>moveOpNCNN requestEntry</td>
<td>New Component on New Node</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveOpNCLN requestEntry</td>
<td>New Component on Database</td>
</tr>
<tr>
<td></td>
<td>Guide</td>
<td>Component</td>
<td>moveCompNN (6.1) requestEntry</td>
<td>GardenStateHandling</td>
</tr>
<tr>
<td></td>
<td>Guide</td>
<td>Component</td>
<td>moveCompLN requestEntry</td>
<td>Greenhouse Management</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reduceUsages (4) getPeopleAmount</td>
<td>Guide</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveOpNCNN isGuideInside</td>
<td>New Component on New Node</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveOpNCLN isGuideInside</td>
<td>New Component on Database</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveOpLC isGuideInside</td>
<td>GardenStateHandling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveCompLN</td>
<td>DatabaseManagement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reduceUsages (4) isGuideInside</td>
<td>Guide</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveCompLN</td>
<td>Database</td>
</tr>
<tr>
<td></td>
<td>Server-Camera</td>
<td>Node</td>
<td>moveOpNCNN checkCredentials</td>
<td>New Component on New Node</td>
</tr>
<tr>
<td></td>
<td>Server-DATABASE</td>
<td></td>
<td>moveOpNCLN checkCredentials</td>
<td>New Component on Database</td>
</tr>
<tr>
<td></td>
<td>Server-Mobile</td>
<td></td>
<td>moveOpLC checkCredentials</td>
<td>GardenStateHandling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveCompLN Security</td>
<td>New Node</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveCompLN</td>
<td>Database</td>
</tr>
<tr>
<td>PaF</td>
<td>checkCredentials</td>
<td>Operation</td>
<td>moveOpNCNN checkCredentials</td>
<td>New Component on New Node</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveOpNCLN checkCredentials</td>
<td>New Component on Database Node</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveOpLC checkCredentials</td>
<td>GardenStateHandling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveCompLN Security</td>
<td>New Node</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>moveCompLN</td>
<td>Database</td>
</tr>
</tbody>
</table>

node #
Figure 12: Refactoring Tree.
• 7 nodes meet both requirements, i.e., 1.1, 1.2, 2, 5.1, 6, 6.1, 6.1.1.

• 4 nodes do not meet both requirements, i.e., 1, 3, 4, 5.

The refactoring tree is representative of the main situations that may occur while refactoring a software model to meet performance requirements. In fact, it shows

• cases where performance requirements are met in one move only, i.e., 2 and 6.

• cases where performance requirements are only partially met, i.e., 1.

• cases where performance requirements are not met but performance indices are better than the ones of the initial model, i.e., 5.

• cases where performance indices diverge, i.e., 3 and 4.

Edges of the refactoring tree.

Each edge connecting two nodes represents the application of a refactoring: in particular, the edge is annotated with the name of the targeted performance antipattern occurrence(s) (see the table below the tree) and a string that codifies the refactoring application.

Edges originating from the root represent the first set of refactoring actions that we have considered, i.e., the ones selected in the refactoring reasoning step (Table 7). We thus have applied the following refactoring actions to the initial BGMS model (see nodes 1-6):

1 Server.moveOpNCNN, targets the corresponding CPS and PaF occurrences. After this refactoring, the PaF occurrence is removed, but the Server is affected by a new CPS occurrence. In particular, the latter occurs between the Server Node and the new UML Node, i.e., ServerC, introduced by such refactoring. Node 1 of the tree does not meet both the performance requirements. However, it is a promising node due to the fact CS meets its performance requirement, whereas AG response time, although not satisfactory, has improved with respect to BGMS node. Hence, node 1 is worth to be further explored.
Server.moveCompNN targets the corresponding CPS and PaF occurrences. Similarly to the previous case, after this refactoring, the PaF occurrence is removed, but the Server is affected by a new CPS occurrence, in particular, between the Server Node and the new UML Node, i.e., ServerC, introduced by such refactoring. However, differently from the previous case, node 2 meets both the performance requirements. Hence, it does not need to be further explored.

GreenhouseManagement.reduceUsage targets the corresponding Blob occurrence. After this refactoring, the Blob occurrence is not removed due to new threshold values for Component connections, which are lower than the initial ones. Also the actual CPS occurrence still affects the system, and an additional PaF occurrence is introduced, i.e., the fetchAccount Operation of the Database Component. This results in a refactored model where AG and CS response times are not only unacceptable, but they are even much worse than the initial ones. Hence, node 3 is not worth to be further explored.

Guide.reduceUsage targets the corresponding Blob-Controller occurrence. Similar arguments of the previous case apply to this one, except for the fact that AG and CS response times are not that bad.

Guide.moveCompNN targets the corresponding Blob occurrence. After this refactoring, the Blob occurrence is removed, but the Server is still affected by an additional CPS occurrence between the Server Node and the new UML Node, i.e., ServerC, introduced by such refactoring. Node 5 does not meet both the performance requirements. However, it can be considered analogous to BGMS node, since AG and CS response times similar to the initial ones (i.e., AG response time is slightly improved, whereas CS one has slightly degraded). Hence, node 5 is worth to be further explored.

Server.moveCompLN targets the corresponding CPS and PaF occurrences. After this refactoring, the PaF occurrence is removed, but the Server is still affected by a CPS occurrence, in particular, between the Server Node and the DB Node. Moreover, two new antipattern occurrences are introduced: A PaF, with respect to fetchAccount Operation, and an EP, with respect to getGardenState (and displayError) Operation(s). The introduction of additional antipatterns is due to the fact
that Security is moved to the less used Node, i.e., DB, which is not the less used among all the UML Nodes of the BGMS but the less used among all the Nodes not excluded by legacy constraints. Node 6 of the tree meets both the performance requirements, hence it should not be worth to be further explored. In fact, we assume this for in case of refactoring sessions with EVL and EWL. However, in case of EPL, things are different since the engine does not stop, thus continuing to detect further antipatterns (see dashed arrows of the refactoring tree). More details are demanded to the remainder of this section.

Stepping down to the deeper layers of the tree, we applied the following refactoring actions:

1.1 By applying Guide.moveCompLN from node 1, the corresponding Blob occurrence is targeted. After this refactoring, the Blob occurrence is removed, but the Server is still affected by the CPS occurrence as before, i.e., between the Server Node and the Database and ServerC Nodes. Node 1.1 of the tree meets both the performance requirements. Hence, it does not need to be further explored.

1.2 Alternatively to node 1.1, by applying Server.moveOpNCLN, from node 1, the corresponding CPS and PaF occurrences are targeted. On the one hand, no antipattern occurrences have been removed. On the other hand, no antipattern occurrences have been introduced. However, node 1.2 meets both the performance requirements. Hence, it does not need to be further explored.

5.1 By applying Server.moveCompLN from node 5, the corresponding CPS and PaF occurrences are targeted. On the one hand, no antipattern occurrences have been removed. On the other hand, no antipattern occurrences have been introduced. However, node 5.1 meets both the performance requirements. Hence, it does not need to be further explored.

6.1 By applying Guide.moveCompNN from node 6, the Blob caused by Guide is targeted. After this refactoring, the Blob occurrence is removed, as well as the previous CPS occurrence affecting the Server,
i.e., Server-DB. However, the Server is still affected by a CPS occurrence as before, but with respect to ServerC, introduced by Guide.moveCompNN. ServerC is also involved in an additional CPS occurrence that has been introduced, affecting the DB. Node 6.1 of the tree meets both the performance requirements and, as in case of its predecessor, it should not be worth to be further explored. However, since they may be intermediate nodes in case of EPL and EVL, we have analyzed its performance. Results show that AG response time improves by 3 seconds circa, whereas CS one remains almost unchanged.

6.1.1 By applying GreenhouseManagement.moveCompNN from node 6.1, the Blob caused by GreenhouseManagement is targeted. As for its predecessor, the Blob occurrence is removed. However, the system is still affected by the same CPS occurrence as before (i.e., Server and DB), with respect to both the UML Nodes that have been introduced from the first refactoring. Node 6.1.1 of the tree meets both the performance requirements and, differently from its predecessors, it is not further explored because it is a final node of a batch refactoring session executed in non-iterative mode.

Refactoring effectiveness.

Several observations concerning refactoring effectiveness can be made with respect to the conducted experimentation, as reported in the following.

- Targeting the CPS occurrence on the initial BGMS implies targeting the PaF occurrence, and vice versa. However, targeting CPS/PaF (i.e., moving Security to a new node) only removes the PaF occurrence and improves the performance of the software system (see nodes 1 and 2). As a result, the demand of Security is either partially (see node 1) or totally (see node 2) moved to a new Node, thus reducing the load on the original Server Node. Nevertheless, the former case is not enough to fulfill the performance requirements, although a new Node is introduced. Hence, further refactoring is necessary in this case, as described in the following. Once part of Security has been moved to a new Node, the latter could be the less used Node, whereas another component (named Guide) is the most demanding Component deployed to the Server Node.
Hence, targeting Guide as a Blob (i.e., by applying Guide.moveOpLN, see node 1.1) or its most demanding Operation as source of a CPS occurrence (i.e., by applying Server.moveOpNCLN, see node 1.2) would lead to the performance requirements fulfillment.

- Node 5 confirms the fact that the introduction of a new node does not necessarily improve the performance, as well as it does not necessarily reduce the number of antipattern occurrences. However, from the tree we observe that a refactored model that does not meet performance requirements generally shows a number of antipattern occurrences larger or equal to the ones occurring in its original model. Similarly to the previous point, once the new Node has been introduced, a second reduction of the Server load that targets a CPS occurrence (i.e., by applying Server.moveCompLN, see node 5.1) would lead to the performance requirements fulfillment.

- reduceUsages action results in worsening performance (see nodes 3 and 4), although it is a specific action for the Blob.

- Results show that node 6.1.1 is an optimum among the considered ones. Hence, the two additional refactoring actions have led to an optimal solution. This has been shown to highlight that a minimalist approach stopping as soon as performance requirements are fulfilled (see ✓ on nodes 1, 2, and 6) would have not led to this point.

**Session support by the three EPSILON engines.**

In this subsection we illustrate how the tree in Figure 12 has been obtained by executing the different sessions that are enabled within our framework (see Figure 4), in order to address RQ2 (see Section 1).

*EPL - Batch refactoring session.*

Since each EPL pattern allows to specify only one imperative block (i.e., a single refactoring action), in order to enable multiple refactoring actions for an antipattern we have to write (or obtain by porting from EVL or EWL) more than one pattern with the same antipattern detection rule and different imperative blocks. We have experienced that this practice introduces
non-determinism in the pattern choice by the EPL engine during the detection phase. For this reason, in our experimentation we enabled only one refactoring for each antipattern, that are detailed in the following:[13]:

1. moveCompLN for CPS - in order to reduce utilization of the Server, which is the overloaded node of a CPS occurrence, while distributing the demand of the most demanding hosted Operation on the less used Node (i.e., DB, after excluding the legacy constrained ones).

2. moveOpLC for PAF - in order to distribute the demand of the Operation that plays the role of slowest filter in the PaF occurrence (i.e., checkCredentials) on the less demanding Component (i.e., Database, after excluding the legacy constrained ones), thus reducing the utilization of the Server Node hosting the slowest filter.

3. moveCompNN for Blob - in order to reduce the utilization of the Server Node, which hosts the Blob occurrence, by distributing its demand on a new Node.

The antipattern ordering in the EPL file is fundamental, as the EPL engine parses EPL patterns as they appear in that file. In our experimentation, antipatterns are ordered as in the list above. As a result, the PaF occurrence is removed after targeting the CPS one by applying moveCompLN, hence moveOpLC is never applied in the batch session (i.e., nodes 6, 6.1, and 6.1.1).

While executing a batch refactoring session, the EPL engine maintains intermediate models resulting from the execution of the antipattern rules in that session. As a side effect, since the refactoring actions operate on design aspects (i.e. static, dynamic and deployment views), the performance indices are never updated within this session, because this update requires the generation and the simulation of a new QN model, which is unfeasible by definition in a batch session.

For example, after the execution of Server.moveCompLN (node 6 of the refactoring tree), the Server in the resulting intermediate model will have the same utilization as the initial model. Instead, since Security has been cut off from the Server, its utilization should decrease possibly implying the removal

[13]Note that since EP, EST, and ToB are never detected at intermediate nodes in our case study, they will not be considered here.
of other antipattern occurrences. However, as evident in the tree, this aspect does not prevent the batch session to end in an optimal model.

Hence, in its current version our framework lacks of an automated update of performance indices of intermediate models. An alternative to the generation of QN models to be simulated would to perform algebraic estimation of these indices [15]. We leave the automated update of performance indices of intermediate models as a future improvement of our work. For sake our experimentation, a performance analysis has been manually conducted for each node of the refactoring tree.

**EVL - User-driven multiple refactoring session.**

We recall that this is a multi-steps session. In fact, it allows to sequentially apply multiple refactoring actions, where each action is chosen by the user at each step among the ones enabled on the basis of detected antipatterns.

As for EPL, the EVL engine maintains intermediate models. The lack of an automated update of performance indices of intermediate models mentioned for EPL still holds for EVL. The difference is that here the user decides when stopping the session, hence she decides when opening the possibility to generate and simulate new QN models for sake of performance indices update.

For example, in Figure 12 we have labeled the transition from the initial model to node 5.1 as such a session.

**EWL - User-driven single refactoring session.**

The EWL engine allows to reach any node of the refactoring tree directly within the modeling editor. Differently from EPL and EVL, no intermediate models are maintained by the EWL engine. In fact, the executed refactoring is finalized step-by-step during the single refactoring session. After such finalization, a performance analysis shall be conducted in order to obtain performance indices.

For example, in Figure 12 we have labeled the transition from the initial model to node 1 as such a session. In general, every single step in the tree can be obtained from the execution of this session.

5.4. Results overview

The experimentation conducted on the BGMS case study has been aimed at providing evidence to the answers of research questions introduced at the beginning of this paper. We have not considered a large-size case study (i.e.,
ten software components and six deployment nodes), because the considered UML model size allowed to point out the details discussed in previous sections, which would have been hardly captured on a wider scale. Of course, the scalability of our approach on industrial case studies is one of our future investigations, but it is out of this paper scope.

In the remainder of this section we summarize the lessons learned with regard to RQ2 and RQ3, because the adequacy of Epsilon to implement detection rules and refactoring actions (i.e., RQ1) has been widely proven by the EPL/EVL/EWL code that we have implemented in our framework.

5.4.1. RQ2 - Refactoring sessions

As outlined before, the choice of EPL, EVL and EWL has enabled three types of sessions that, as also illustrated in Figure 12, open different scenarios on the tradeoff between automation and human participation to the refactoring process.

Even though in Figure 12 we have checked the requirement satisfaction for intermediate models along the rightmost batch session, in practice these models are not visible to the framework users. In fact, this session applies antipattern detection and refactoring to the whole model without the human intervention, because it is aimed to push the refactoring as more ahead as possible, even if in our case the objective was reached before. This session applies to scenarios where optimal models are pursued in absence of specific performance requirements.

At the opposite extreme of the session range (but also by chance in the figure!), in the leftmost user-driven single session in Figure 12 users intend to proceed step-by-step in the refactoring process by applying actions on single model elements that, after a local detection step, have been identified as involved in some performance antipattern. This session applies to scenarios where minimal local refactoring are pursued under the human supervision.

Finally, the user-driven multiple session highlighted on the right side of Figure 12 is conducted on a specific console, where multiple user-selected refactoring actions are applied to the whole model. This session applies to all intermediate scenarios between the above ones.

Table 8 summarizes the classification of introduced sessions along the application scope and the automation level.
Table 8: Summary of sessions enabled from our framework.

<table>
<thead>
<tr>
<th>Sessions</th>
<th>Automation Level</th>
<th>Application Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch - EPL</td>
<td>High</td>
<td>Model</td>
</tr>
<tr>
<td>Multiple User-Driven - EVL</td>
<td>Medium</td>
<td>Model</td>
</tr>
<tr>
<td>Single User-Driven - EWL</td>
<td>Low</td>
<td>Element</td>
</tr>
</tbody>
</table>

5.4.2. RQ3 - Performance improvement

For many years, traditional bottleneck analysis [44] has been the main tool in the hand of software engineers for detecting performance problems. However, bottleneck analysis cannot capture fine-grain causes of performance degradation, which possibly involve aspects belonging to different model views. As we have shown in [12], performance antipatterns can compensate this limitation because they embed such different aspects in their definition.

Our first consideration in the context of RQ3 is related to the tightness of performance requirements to primary analysis results. The requirements that we have considered for BGMS are reported on top of the refactoring tree in Figure [12] whereas the primary results obtained from analyzing the model as-is are reported under the tree root. We note that the distance between requirements and results is not huge, i.e. about 25% and 5% respectively.

As demonstrated in our experimentation, this is a typical case where performance antipatterns offer a variety of refactoring possibilities to improve performance indices (i.e., six different paths departing from the tree root), even though the distance between indices and requirements is not large.

Altogether these paths originate a tree with 11 nodes, each representing a different model obtained from refactoring the original BGMS one in the root. By discarding nodes 6 and 6.1 that, as said above, are not actually visible to the framework users, 4 out of 9 nodes do not lead to requirement satisfaction. By rough quantification, around 44% of the refactoring effort, including modeling and antipattern detection, has been useless. This (apparently!) negative balance can be afforded only if most part of this effort is assigned to automated tools like the one at the basis of our framework.

As a compensation of such effort, five different refactoring solutions have been proposed to software engineers in order to satisfy the performance requirements. Additional considerations, for example related to costs of refactoring actions, could drive the choice of the solution to take.

Finally we remark that, with the appropriate support, the paths leaving the refactoring tree of Figure [12] can be run in parallel in order to improve the scalability of the approach mentioned above.
6. Threats to Validity

The threats to validity of our experimentation are discussed in the following, according to the macro-categorization given by Wohlin et al. [47]. Note that most of them are internal threats to validity, due to assumptions made in the current version of our framework.

**Conclusion validity.** *It concerns the assessment of a significant relationship between the treatment and the obtained results.*

Such type of threat arises in the coverage of the solution space, because a restricted exploration may lead to non-optimal results, i.e. satisfactory solutions could not be found. In this regard, exploring the whole solution space (i.e., applying all the possible refactoring actions for each node of the refactoring tree) would have been unfeasible. Hence, we had to limit the considered actions to a subset of the whole portfolio. We have made this selection in a controlled way, by choosing a subset leading to a representative refactoring tree, i.e., a tree that includes heterogeneous nodes and refactoring actions. This could have neglected promising refactoring actions, but the goal of the experimentation was to illustrate the framework effectiveness and not its optimality.

**Internal validity.** *It concerns the causal relationship to assess between the treatment and the outcome of the experimentation, which has not to be a result of non-controllable or non-measured factors.*

Due to the complexity of QN models, we had to simulate them in order to get performance indices. Hence, we incurred in the usual simulation threats related to confidence intervals. However, the accuracy of obtained performance indices can threaten the validity of our approach only in borderline cases, where these values are very close to performance requirements. For example, this is the case of node 5 of the refactoring tree in Figure 12. The decision whether stopping the process or not is affected from the simulation accuracy. Of course, this risk can be mitigated by pushing the simulation accuracy further, at the cost of longer simulation time.

Moreover, UML models in input to our framework, as specified in Section 4.1, are expected to be complete. This is not only preliminary required by the **Performance-oriented Model Integration** step of Figure 2 (e.g. for performance model generation), but also by the EPL/EWL/EVL codes that implement antipattern detection rules and refactoring actions. Hence, in its current implementation, the framework cannot tolerate any model incompleteness, despite the fact that often software engineers, in practice, have to
deal with incomplete models. This threat seems complex to mitigate with respect to the **Performance-oriented Model Integration** step, where consolidated techniques are based on complete models. Instead, it could be mitigated on the side of EPL/EWL/EVL rules. The latter can in fact be adapted to a set of pre-specified model incompletenesses, where obviously the result of their application will be a partial detection of antipatterns. Similarly, partial refactoring actions can be devised \[^{[14]}\].

Finally, the lack of automation in filling the intermediate models processed by EVL and EPL with updated performance indices may lead these two engines to loose accuracy, as for example:

- At node 6 of the refactoring tree in Figure 12, a session based on EPL is not aware of the fact that the performance of the initial model have been improved by the applied refactoring (i.e., Server.moveCompLN). Hence, the EPL engine keeps refactoring the model, because it refers to the initial one. For this reason, EPL might incur in very long paths in the tree (not shown in Figure 12), especially when working in iterative mode. In some cases, this might result in a final model with worse performance than the initial one, even with more antipatterns.

- The same arguments apply to EVL but, differently from EPL, during a refactoring session with the former, the designer chooses if she either wants to stop the session after a refactoring application or going ahead by applying a further refactoring. Note that, in this latter case, the designer might choose to target an antipattern among the detected ones. Hence, EVL is affected by the same threat described above for EPL, but it is mitigated by the designer assistance during the refactoring session.

**Construct validity.** *It concerns generalizing experiment results to the concept or theory behind the experiment.*

Our experimentation is aimed at showing that performance antipatterns represent powerful instruments in the hands of developers and performance experts that can be exploited to improve performance of software systems,

\[^{[14]}\]In our previous work \cite{[15]} , we have presented an approach for performance antipattern detection and refactoring under uncertainty on the values of thresholds occurring in the formulation of antipatterns, where we have introduced the concepts of antipattern probability and refactoring benefits that could be reused in a context of model incompleteness.
which in this paper corresponds to $RQ_3$. Despite the internal threats highlighted above, we have shown that our antipattern-based framework is able to improve performance of an existing software system and fulfill the predefined performance requirements. However, as outlined in Section 7 extensive validation on a large number of models would be needed in order to identify the cases where our approach works better. Such extensive validation shall be conducted after a complete implementation of the Performance-oriented Model Integration step of Figure 2 that has to extend the EVL and EPL engines with the aim of filling the intermediate models with updated performance indices. In fact, the lack of automation in this step not only affects internal validity of our experimentation, but also its construction validity with respect to theory, because the reference to performance indices of the initial model may threaten precision and recall of performance antipatterns detection and, consequently, it may mislead solution space exploration in the worst case.

**External validity.** It refers to the generalizability of obtained results to industrial practice.

So far, we have experimented the effectiveness of our framework with respect to UML models properly profiled with MARTE, which are quite widely used in industry, thus mitigating threats to external validity. However, beyond UML, several industrial contexts tend to define their own domain-specific languages, which obviously are not currently supported by our framework. For this reason, as outlined in Section 7, we intend to demonstrate the validity of the support provided by our framework over different domain-specific metamodels. This appears as a complex and time-consuming task, but Model-Driven Engineering techniques such as weaving models might be exploited for this goal.

7. Conclusion

In this paper we have introduced a framework for automation of software model refactoring driven by performance issues. At the best of our knowledge, this is the first work that aims at managing the performance-driven evolution of a software model within an unique environment, i.e. ESPILON in our case.

Our experimentation demonstrates that the framework provides a very valid support to drive UML model refactoring towards models that satisfy performance requirements. Without such support, software designers and
performance experts should rely only on their own experience to find performance problems and related solutions.

In Section 6 we have discussed the major threats to validity of our approach, which we intend to address in the near future. Beside those issues, however, a large scale (possibly industrial) experimentation not only would possibly give evidence to issues that are undetectable in lab, but it would also consolidate the achievements presented in this paper.

As a long-term goal, we intend to demonstrate the validity of the support provided by our framework over different metamodels. The EPSILON code for performance antipatterns detection and model refactoring shall be modified in order to target the modeling constructs of the new metamodels.

References


[16] D. Arcelli, V. Cortellessa, D. Di Pompeo, Towards a Unifying Approach for Performance-Driven Software Model Refactoring,


Appendix A. Performance Antipatterns

In this Appendix we describe in details the six performance antipatterns reported in Table 4. For each of them we describe the problem and possible solutions (in terms of specific refactoring actions for removing it).

Moreover, as we said in Section 3, we have made some changes to performance antipatterns formulae with respect to those published in [8]. Here we describe those changes that we have also marked in bold in Table A.9. In details:

- **Blob.** We have split it in two versions: (i) Blob-controller, and (ii) Blob-dataContainer. Basically, this splitting allows us to identify and better recognize the instances of this antipattern in order to apply appropriate refactoring action(s).

- **Concurrent Processing System.** We have changed the original formula by introducing: \( \exists i \in \mathbb{N}, F_{resDemand}(OpI)[i] \geq Th_{maxOpResDemand}[i] \). Previously, the occurrence of CPS referred to overloaded nodes only, hence by introducing the new clause we can recognize CPS occurrences if and only if the software system under analysis has at least an Operation that shows at least a resource demand greater than relative threshold. By doing so, the detection of this performance antipattern is more precise than before.

- **Extensive Processing.** We have relaxed the formula by introducing: \( \exists i \in \mathbb{N} \) instead of \( \forall i \in \mathbb{N} \). By doing so, the detection of EP is more precise to recognize Operation(s) that generates the performance antipattern. In fact, previously, EP occurrence(s) might not be detected because it was difficult to find systems whose Operation(s) shows every resource demand greater than the relative threshold.

Appendix A.1. Blob Controller and Blob Data Container

**Problem** - Occurs when a single component either i) performs the most part of the work of a software system or ii) holds the most part of the data of the software system. Either manifestations result in excessive message traffic that may degrade performance.

The former is called Blob-controller antipattern, and it is showed in Figure A.13, the latter, instead, is called Blob-dataContainer antipattern, and
Table A.9: A logic-based representation of Performance Antipatterns.

<table>
<thead>
<tr>
<th>Performance Antipattern</th>
<th>Logic Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob (controller)</td>
<td>( \exists c_x, c_y \in C, S \in S \mid F_{numClientConnects}(c_x) \geq T_{maxConnects} \land F_{numMsgs}(c_x, c_y, S) \geq T_{maxMsgs} \land (T_{maxHWUtil}(P_{cy}, all) \geq T_{maxHWUtil} \lor T_{maxNetUtil}(P_{cx}, P_{cy}) \geq T_{maxNetUtil}) )</td>
</tr>
<tr>
<td>Blob (dataContainer)</td>
<td>( \exists c_x, c_y \in C, S \in S \mid F_{numSupplierConnects}(c_x) \geq T_{maxConnects} \land F_{numMsgs}(c_x, c_y, S) \geq T_{maxMsgs} \land (T_{maxHWUtil}(P_{cy}, all) \geq T_{maxHWUtil} \lor T_{maxNetUtil}(P_{cx}, P_{cy}) \geq T_{maxNetUtil}) )</td>
</tr>
<tr>
<td>Concurrent Processing Systems (CPS)</td>
<td>( \exists P_x, P_y \in P, OpI \in \emptyset, i \in N \mid F_{maxQL}(P_x) \geq T_{maxQL} \land ([F_{maxHWUtil}(P_x, cpu) \geq T_{maxCPUUtil} \land F_{maxHWUtil}(P_y, cpu) &lt; T_{minCPUUtil} \lor (F_{maxHWUtil}(P_y, io) \geq T_{maxDiskUtil} \land (F_{maxHWUtil}(P_y, io) &lt; T_{minDiskUtil}))]) \land F_{ResDemand}(OpI)[i] \geq T_{maxOpResDemand[i]} )</td>
</tr>
<tr>
<td>Unbalanced Processing (PaF)</td>
<td>( \exists OpI \in \emptyset, S \in S, i \in N \mid F_{ResDemand}(OpI)[i] \geq T_{resDemand[i]} \land F_{probExec}(S, OpI) = 1 \land (F_{maxHWUtil}(P_c, all) \geq T_{maxHWUtil} \lor F_T(S) &lt; T_{thReq}) )</td>
</tr>
<tr>
<td>Extensive Processing (EP)</td>
<td>( \exists OpI_1, OpI_2 \in \emptyset, S \in S, i \in N \mid F_{resDemand}(OpI_1)[i] \geq T_{resDemand[OpI_1][i]} \land F_{resDemand(OpI_2)[i]} \land F_{probExec}(S, OpI_1) + F_{probExec}(S, OpI_2) = 1 \land (F_{maxHWUtil}(P_c, all) \geq T_{maxHWUtil} \lor F_T(S) &gt; T_{thReq}) )</td>
</tr>
<tr>
<td>Empty Semi Trucks (EST)</td>
<td>( \exists c \in C, S \in S \mid F_{numRemMsgs}(c, S) \geq T_{maxRemMsgs} \land F_{numRemInst}(c, S) &lt; T_{minNetUtil} \lor F_{numRemInst}(c, S) \geq T_{maxHWUtil} \land F_{maxHWUtil}(P_c, all) \geq T_{maxHWUtil} )</td>
</tr>
<tr>
<td>Tower of Babel (ToB)</td>
<td>( \exists c \in C, S \in S \mid F_{numExF}(c, S) \geq T_{maxExF} \land F_{maxHWUtil}(P_c, all) \geq T_{maxHWUtil} )</td>
</tr>
</tbody>
</table>
it is described in Figure A.14. The performance impacts of both cases are mainly due to the consequent excessive message passing among the blob component and the other ones. The performance loss is clearly heavier on distributed systems, where the time needed to pass data among remote components is significant with respect to the computational time.

The upper side of Figures A.13 and A.14 describes the properties of a Software Model M with a Blob problem: (a) Static View, a component $C_x$, is connected to another one namely $C_y$, through many dependencies; (b) Dynamic View, $C_x$ generates (case of Blob-controller see Figure A.13) or receives (case of Blob-dataContainer see Figure A.14) excessive message traffic to elaborate data managed by $C_y$; (c) Deployment View, it includes two subcases: (c1) the centralized case, where the communicating components are deployed to the same node $N$, which shows high utilization $\text{util}$; (c2) the distributed case, where the communicating components are deployed to different nodes, and the connecting network link might be a critical resource with a high utilization $\text{utilNet}$. The occurrence of such properties leads to assess that the component $C_x$ originates an instance of the Blob antipattern.

**Solution** - Refactor the design to distribute intelligence uniformly over the software system’s top-level classes, and to keep related data and behaviour together, by increasing the cohesion and reducing the coupling of the software system.

The lower side of Figures A.13 and A.14 shows the design changes that can be specifically applied aimed at removing a Blob. The following refactoring actions are represented: (a) the number of dependencies between $C_x$ and $C_y$ are decreased by delegating some functionalities to other components; (b) the number of messages sent (see Figure A.13) or received (see Figure A.14) by $C_x$ are decreased by removing the management of data belonging to other components. As consequences of previous actions: (c1) if the communicating components were deployed on the same node then the latter should not be a critical resource anymore, i.e. $\text{util}^\prime < \text{util}$; (c2) if the communicating components are deployed on different nodes then the network link should not be a critical resource anymore, i.e. $\text{utilNet}^\prime < \text{utilNet}$.

**Appendix A.2. Concurrent Processing System (CPS)**

**Problem** - Occurs when processing cannot make use of available processors.

The antipattern, which is described in Figure A.15, represents a manifestation of the Unbalanced Processing antipattern [37]. It occurs when processes cannot make effective use of available processors either because of
Figure A.13: A graphical representation of the Blob-controller Antipattern.
Figure A.14: A graphical representation of the Blob-dataContainer Antipattern.
i) a non-balanced assignment of tasks to processors or because of ii) single-threaded code. In the following we only consider the former, since the application code is an abstraction level typically not included in architectural models.

The upper side of Figure A.15 describes the system properties of a Software Model M with a CPS problem: (a) Deployment View, there are two nodes $N_{over}$ and $N_{under}$, many tasks from components $A_a$, $A_b$, $A_c$ are assigned to $N_{over}$ whereas $N_{under}$ is not so heavily used (e.g. computation of the component $A_d$). The overloaded node shows high queue length value, i.e. $ql_{1}$, and a high utilization among its hardware entities either for cpus, e.g. $cpu_{1}$, and disks, e.g. $disk_{1}$. The less used node, instead, shows low utilization among its cpus, e.g. $cpu_{2}$, and-or disks, e.g. $disk_{2}$, devices. The occurrence of such properties leads to assess that the node(s) $N_{over}$ (and $N_{under}$) originates an instance of the CPS antipattern.

**Solution** - Restructure software or change scheduling algorithms to enable concurrent execution.

The lower side of Figure A.15 shows design changes that, if applied, might lead to solution. Basically, the critical component, i.e. the most demanding one over the components deployed to the overloaded node, must be deployed to a non-heavly utilized node, accordingly to the available nodes or by introducing a new one.

Appendix A.3. Pipe and Filter (PaF)

**Problem** - Occurs when the slowest filter in a "pipe and filter” causes the system to have unacceptable throughput.

The PaF antipattern represents a manifestation of the Unbalanced Processing antipattern [37]. It occurs when the throughput of the overall system is jeopardized by the slowest filter. This is due to a stage in a pipeline, which is significantly slower than all the others, therefore constituting a bottleneck in the whole process in which most stages have to wait the slowest one to terminate.

The upper side of Figure A.16 describes the system properties of a Software Model M with a PaF problem: (a) Static View, there is a component $C$ that provides an Operation $Op$ whose resource demand (computation = $compOp$, storage = $storOp$, and bandwidth = $bandOp$) is high; (b) Dynamic View, the Operation $Op$ is invoked in an interaction $S$ showing a low throughput ($Th(S)$); (c) Deployment View, the node $N$ which $C$ is deployed to, might have a high utilization value ($util$). The occurrence of
Figure A.15: A graphical representation of the Concurrent Processing Systems Antipattern.
"Pipe and Filter Architectures" problem

Figure A.16: A graphical representation of the Pipe and Filter Antipattern.
such properties leads to assess that the Operation $Op$ originates an instance of the PaF antipattern.

Solution - Move the heaviest Operation in a new component that is deployed to a new node in order to remove the bottleneck in the pipe.

The lower side of Figure A.16 shows design changes that, if applied, might lead to the PaF solution: (a) the Operation $Op$ is moved to a new component, namely $C_{new}$; and (b) the latter is deployed to a new node, namely $N_{new}$.

Appendix A.4. Extensive Processing (EP)

Problem - Occurs when extensive processing in general impedes overall response time.

The EP antipattern represents a manifestation of the Unbalanced Processing antipattern [37]. It occurs when a long running process monopolizes a processor and prevents a set of other jobs to be executed until it finishes its computation. The processor is removed from the pool, but unlike the pipe and filter, other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload.

The upper side of Figure A.17 describes the system properties of a Software Model $M$ with a EP problem: (a) Static View, there is a component $C$ that provides two Operations $Op_x$ and $Op_y$ whose resource demand is unbalanced. $Op_x$ has a high demand (computation = $compOp_x$, storage = $storOp_x$, bandwidth = $bandOp_x$), whereas $Op_y$ has a low demand (computation = $compOp_y$, storage = $storOp_y$, bandwidth = $bandOp_y$); (b) Dynamic View, $Op_x$ and $Op_y$ are alternatively invoked in an interaction $S$, and the response time ($RT(S)$) of the latter is too high; (c) Deployment View, the node which $C$ is deployed to, namely $N$, should reveal a high utilization $util$. The occurrence of such properties leads to assess that the Operation(s) $Op_x$ (and $Op_y$) originates an instance of the EP antipattern.

Solution - Move extensive processing so that it does not impede high traffic or more important work.

A solution to the EP antipattern is to identify processing steps that may cause slow downs and delegate those steps to processes that will not impede the fast path. A performance improvement could be achieved by delegating processing steps which do not need a synchronous execution to other processes.

The lower side of Figure A.17 shows the design changes that, if applied, might lead to an EP solution: (a) the Operations $Op_x$ and $Op_y$ are provided
"Extensive Processing" problem

Software Model M

(a) Static View

Software Model M'

(a) Static View

"Extensive Processing" solution

Figure A.17: A graphical representation of the Extensive Processing Antipattern.
by two different components namely \( C \) and \( C_{\text{new}} \) (i.e. an additional one), respectively; (b) \( C \) and \( C_{\text{new}} \) are deployed to different nodes \( N \) and \( N_{\text{new}} \), in order to provide a fast path for requests \( [48] \). As a consequence of the previous actions, the response time of the interaction \( S \) is expected to improve, i.e. \( RT(S)' < RT(S) \).

Appendix A.5. Empty Semi-Trucks (EST)

**Problem** - Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.

The problem of inefficient use of available bandwidth typically affects message-based systems when a huge load of messages, each containing a small amount of information, is exchanged over the network. The amount of processing overhead is the same regardless of the size of the message. With smaller messages, this processing is required many more times than necessary, hence it significantly implies a performance loss. The problem of an inefficient interface (i.e. it provides a too fragmented access to data) generates an excessive overhead caused by the computation needed to handle each call request.

The upper side of Figure A.18 describes the system properties of a Software Model \( M \) with an EST problem: (a) Static View, there is a component, namely \( C \), using some Operations provided by a remote component, namely \( C_{\text{rem}} \); (b) Dynamic View, the component \( C \) generates an excessive message traffic by sending a high amount of messages with light payloads \( \text{msg}_S \), much lighter than the network bandwidth; (c) Deployment View, the node which \( C \) is deployed to, namely \( N \), reveals a high utilization \( util \). The occurrence of such properties leads to assess that the component \( C \) originates an instance of the EST antipattern.

**Solution** - The Batching performance pattern \([48]\) and Session Facade design pattern \([49]\) combine light messages into a heavier message to make better use of available bandwidth. The Coupling performance pattern \([48]\), and Aggregate Entity design pattern \([50]\) provide more efficient interfaces.

In the case of inefficient use of available bandwidth the solution is given by the adoption of either Batching performance pattern \([48]\) that basically groups information in larger chunks in order to minimize the overhead due to information spread and processing, or Session Facade design pattern \([49]\). The latter applies several architectural changes: two components (i.e. LocalFacade and RemoteFacade respectively) have to be created, the former is
deployed in the same node of the sending component, i.e. \( N \), and the latter in the same node of the recipient component, i.e. \( N_{\text{rem}} \). In both solutions (i.e. Batching and Session Facade) several messages are merged into a single heavier message, in order to reduce the access to the network. In the case of inefficient interface a solution could be achieved through the implementation of the Coupling performance pattern \([48]\) that basically uses more coarse-grained objects in order to reduce the amount of communication overhead required to obtain data.

The lower side of Figure A.18 shows the design changes for applying the Session Facade design pattern \([49]\), according to one of the possible specific EST solutions: the communication between \( C \) and \( C_{\text{rem}} \) is restructured, by introducing LocalFacade, which provides a mirroring of the Operations of \( C_{\text{rem}} \) and compresses messages sending by \( C \) in one message only that is subsequently sent through the net to the RemoteFacade component, which decompresses it and delivers to \( C_{\text{rem}} \).

Appendix A.6. Tower of Babel (ToB)

**Problem** - Occurs when processes use different format of data and they spend too many times in convert them to an internal format.

The ToB antipattern occurs in complex, distributed data-oriented systems in which the same information is often translated into an exchange format (by a sending process) and then parsed and translated into an internal format (by receiving processes). The performance loss in this case is clearly due to the excessive overhead caused by the translation and parsing Operations which may be executed several times in the whole execution process.

The upper side of Figure A.19 describes the system properties of a Software Model \( M \) with a ToB problem: (a) Static View, there are some components, namely \( C_x, \ldots, C_i, \ldots, C_n \); (b) Dynamic View, the component \( C_x \) performs too many times the format translation for communicating with other components; (c) Deployment View, the node which \( C_x \) is deployed to, namely \( N \), reveals a high utilization \( \text{util} \). The occurrence of such properties leads to assess that the component \( C_x \) originates an instance of the ToB antipattern.

**Solution** - If possible, the most common format may be adopted. The Fast Path performance pattern \([48]\) identifies paths that should be streamlined. Minimize the conversion, parsing, and translation on those paths by using
“Empty Semi Trucks” problem

Software Model M

(a) Static View

(b) Dynamic View

(c) Deployment View

“Empty Semi Trucks” solution

Software Model M’

(a) Static View

(b) Dynamic View

(c) Deployment View

“Empty Semi Trucks” solution

Figure A.18: A graphical representation of the Empty Semi Trucks Antipattern.
the Coupling performance pattern to match the data format to the usage patterns.

A performance improvement can be achieved by deciding a common format, which minimizes the Operations of translation along the core. In simpler cases, a good solution could be easily found by avoiding unnecessary translations among components, typically introduced to adopt standard exchange languages, even when not necessary.

The lower side of Figure A.19 shows the design changes that can be applied according to a specific Tower of Babel solution. The communication between $C_x$ and the other components can be restructured by setting the format to the most common in the interaction (i.e. $\text{format}_i$). As a consequence the utilization of the node hosting $C_x$ is expected to improve, i.e. $\text{util'} < \text{util}$.
"Tower of Babel" problem

Figure A.19: A graphical representation of the Tower of Babel Antipattern.