Towards a Unifying Approach for Performance-Driven Software Model Refactoring

Davide Arcelli, Vittorio Cortellessa, Daniele Di Pompeo

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila
67100 L'Aquila, Italy
`{davide.arcelli|vittorio.cortellessa}@univaq.it`,
`dipompeodaniele@gmail.com`

Abstract. Performance is a pervasive quality attribute of software systems. Since it plays a key role in the success of many projects, it is important to introduce approaches aimed at satisfying performance requirements from the early phases of software life-cycle. However, this is a complex problem, because a large gap exists between performance analysis results and the feedback expected by software designers. Some approaches proposed in the last few years aim at reducing such gap, based on automated Model-Driven Engineering techniques, but they are fragmented across different paradigms, languages and metamodels.
The goal of this paper is to work towards an approach that enables performance problems detection and solution within an unique supporting environment. We rely on the Epsilon platform, which provides an ecosystem of task-specific languages, interpreters, and tools for MDE. We describe the approach that we are implementing, and we show how some of these languages nicely fit the needs of a unifying paradigm for performance-driven software model refactoring.

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

1 Introduction

Over the last decade, research has highlighted the importance of integrating performance analysis in the software development process. Performance is a crucial quality attribute of many software systems, but it is a complex and pervasive property difficult to study. If performance targets are not met, then a variety of negative consequences (such as user unsatisfaction, lost income, etc.) can impact on significant parts of a project [1]. These factors motivate the activities of modeling and analyzing performance of software systems early in the life-cycle, by reasoning on predictive quantitative results.

Despite the amount of model-based performance analysis techniques that have been proposed in the last few years, the identification of performance problems is still critical, because the results of performance analysis (i.e., mean values, variances, and probability distributions) are difficult to be interpreted and translated into software feedback (i.e., design alternatives) improving system performance.

Figure 1 illustrates a typical model-based performance analysis process aimed at detecting and removing performance problems. Shaded boxes represent process activities, whereas white boxes represent artifacts.

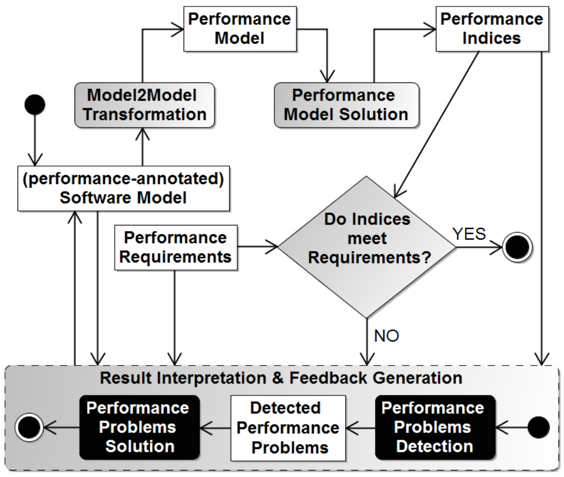

Performance Model
Performance Indices
Model2Model Transformation
Performance Model Solution
(performance-annotated) Software Model
Do Indices meet Requirements?
Performance Requirements
YES
NO
Result Interpretation & Feedback Generation
Performance Problems Solution
Detected Performance Problems
Performance Problems Detection

Fig. 1: Model-based Performance Analysis Process.

The process starts with an initial *(performance-annotated) Software Model*, which represents an abstraction of the system under analysis. Performance annotations are meant to add specific information necessary for performance analysis, such as the incoming workload, service demands, etc. A *Performance Model* (e.g., Petri Net, Queuing Network) is obtained through model transformation from a performance-annotated software model, and it is solved analytically (in most of the cases) or by simulation during the *Performance Model Solution* step, which carries out *performance indices* of interest, e.g., mean response times, throughput distributions, and utilizations.

This represents the forward path of the process. Well-founded model-driven approaches have been introduced in this path for inducing automation in all steps [2], whereas there is a clear lack of automation in the backward path, namely the *Result Interpretation & Feedback Generation* box that shall bring the analysis results back to the software model in order to build a new software model showing better performance.

The backward path takes place in case performance indices do not meet the requirements. In such case performance analysis results have to be interpreted in order to detect *performance problems*. Then solutions (i.e., model refactoring) have to be applied to remove problems.

The contribution of this paper concerns the backward path of the process, and it consists of a first step towards building an approach that enables performance problems detection and solution within an unique supporting environment [3]. To this aim, we rely on some task-specific languages of the Epsilon Platform [4] to implement the *Result Interpretation & Feedback Generation* step. In particular, we exploit their main characteristic, i.e., the possibility of defining (i) *declarative conditions* that can be used to notify the user about performance problems occurrences, and (ii) *imperative blocks* that support software model refactoring based on violations of those conditions.

The paper is organized as follows. Section 2 provides a background on the main ingredients of this work, i.e., performance antipatterns and the Epsilon platform. Section 3 reviews related work. Section 4 represents the core of this paper, that is how to exploit the Epsilon platform to achieve our goal, and finally Section 5 concludes the paper.

2 Background

In this section we provide the background of this work, i.e. *Performance Antipatterns* and the *Epsilon* platform.

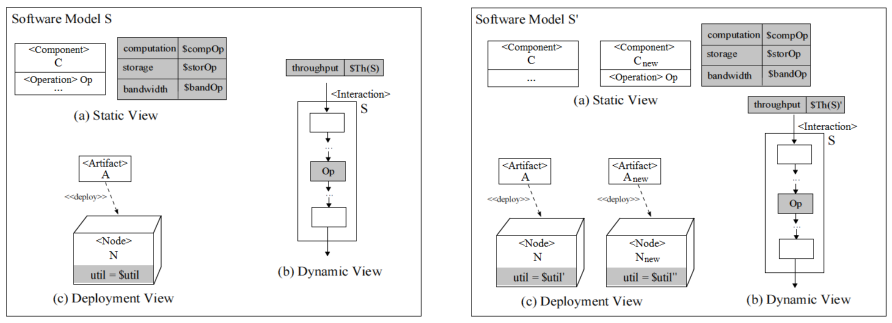
2.1 Performance Antipatterns

Since more than a decade, *Performance Antipatterns* [5,6] revealed to be strong support for performance-driven software model refactoring, since they have been used to: (i) "codify" the knowledge and experience of analysts by describing potentially bad design patterns that have negative effects on performance of software systems, and (ii) remove such negative effects through refactoring actions.

Performance antipatterns have been originally defined in natural language [7,8,9]. In [10] a formal technology-independent interpretation has been provided, based on first-order logic rules, that defines a set of system properties under which a performance antipattern occurs in a software model. A specific characteristic of performance antipatterns is that they contain numerical parameters that represent *thresholds* referring to either performance indices (e.g., high device utilization) or design features (e.g., many interface operations) [11,12].

Figure 2 provides an UML-like graphical representation of the *Pipe & Filter* (PaF) antipattern, which occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput. In Figure 2a, the slowest filter is represented by the Operation Op and its execution causes a bottleneck in the Interaction S, which shows a throughput lower than a certain threshold. This is due to the Operation Op, owned by a Component C, that has resource demands (computation, storage, bandwidth) larger than corresponding thresholds. The Component C is manifested by an Artifact A which is deployed to a Node N showing a mean utilization $util$ which is greater than a certain threshold.

A solution to a PaF antipattern occurrence (see Figure 2b) consists of moving the slowest filter to an ad-hoc software component deployed to 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 2b, the Operation Op representing the slowest filter is moved to a new Component C_{new}, which is manifested by a new Artifact A_{new} deployed on a new Node N_{new}.



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

Fig. 2: Pipe and Filter performance antipattern characterization.

2.2 Epsilon

Epsilon stands for *Extensible Platform of Integrated Languages for mOdel maNagement*. It is a platform for building consistent and interoperable task-specific languages for model management tasks such as model transformation, code generation, model comparison, merging, refactoring and validation.

Epsilon currently provides eigth languages, and for each language Eclipse-based development tools and an interpreter that can execute programs written in that language are provided.

The Epsilon ecosystem provides 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 (i.e *guard* and/or *check*) and imperative blocks (i.e., *do*) that have to be executed if the conditions are not satisfied. This paradigm matches with the concept of detection and solution of performance antipatterns: the declarative conditions codifies antipatterns, whereas in the imperative blocks refactoring actions that might lead to antipatterns solution are codified. The different execution semantics of the three languages mentioned above allow to provide different automated support to the user (see Section 4).

3 Related Work

The problem of model (and metamodel) refactoring has been largely investigated in the last few years, and the refactoring criteria span from usability through modifiability up to evolution needs [13]. However, few approaches deal with performance-driven model refactoring.

The work in [14] presents the ArchE framework that 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.

In [15] a technique for automatic refactoring a Service-Oriented Architecture (SOA) design model by applying a design pattern and for propagating the incremental changes to its Layered Queuing Network (LQN) performance model is introduced. In this paper we take the opposite direction, in that we refactor software models by removing performance antipatterns rather than introducing patterns. In addition, we work for a unifying approach that is not bounded to any paradigm (such as SOA) or performance model (such as LQN).

In [16], two specific EMF-based languages are integrated for model refactoring. Due to the multi-view nature of performance antipatterns, we target models describing software components, their interactions, and their deployment, whereas the approach in [16] mainly considers classes and their interactions, i.e. models that are closer to the code. Moreover, in [16] only design quality attributes and metrics are taken into account, whereas we target performance, i.e. a pervasive software quality attribute that involves more unpredictable metrics.

4 Refactoring Approach

In this section, we describe our approach in some detail. The main idea is illustrated in Figure 3. The approach is centered on three performance antipatterns detection and solution engines, providing different interactive support to the designer, based on EPL, EVL, and EWL. The designer selects the engine to use in order to perform refactoring sessions starting from an initial software model, i.e. M_0, which conforms to a metamodel MM and is given as input to the selected engine. During a refactoring session, a number of new refactored models, i.e., $M_1, .., M_{n-1}$, are created, until a software model that satisfies performance requirements is obtained, i.e. M_n. Note that, for each engine, with respect to the metamodel MM which the software model conforms to, the performance expert has to build the basic knowledge, i.e. K_{MM}, manually. This means that she has to write the EPL/EVL/EWL code that implements performance antipatterns occurring conditions and refactorings that can be applied to remove them.

4.1 Sample Implementation for the PaF Antipattern

Figures 4a, 5a, and 6a, show examples of the PaF detection condition and refactoring of Figure 2, 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., *PaF_resDemand()*, *PaF_F_probExec()*, *PaF_F_throughtput()*, and *PaF_F_maxHwUtil()*) embedded in the PaF formulation [10].

The *do* block of a pattern/critique/wizard contains a call to the operation named *moveToNewComponentDeployedOnNewNode*, which codifies the PaF refactoring illustrated in Figure 2b[1].

[1] The implementation can be found at `http://www.di.univaq.it/davide.arcelli/resources/projects/Epsilon4PAs.rar`.

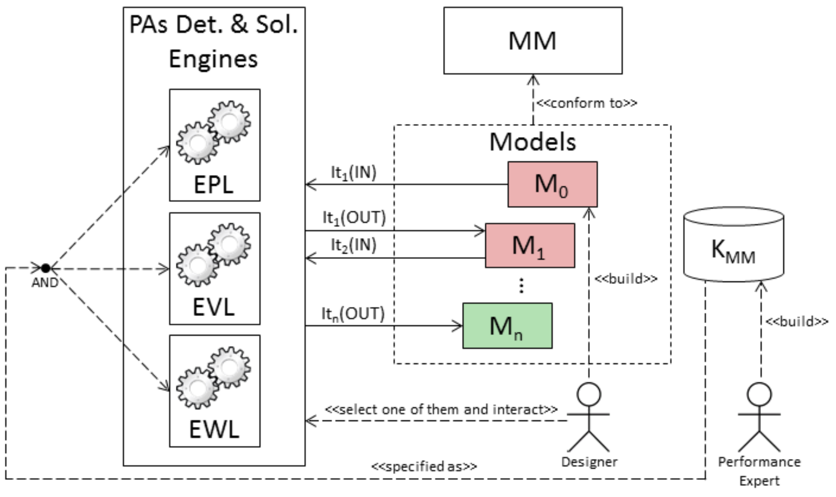
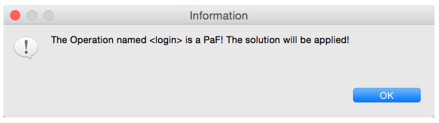


Fig. 3: Epsilon-based approach for model refactoring.

```
pattern PaF
    mainRole : Operation {
    match :
        mainRole.PaF_resDemand() and
        mainRole.PaF_F_probExec() and
        (mainRole.PaF_F_maxHwUtil() or mainRole.PaF_F_throughtput())
    do {
        mainRole.moveToNewComponentDeployedOnNewNode();
    }
}
```

(a) Excerpt of an EPL pattern for PaF.



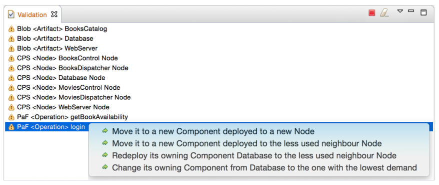
The Operation named <login> is a PaF! The solution will be applied!

(b) Refactoring session with EPL.

Fig. 4: EPL refactoring engine.

```
context Operation {
    critique PaF {
        check {
            if(self.PaF_resDemand() and
               self.PaF_F_probExec() and
               (self.PaF_F_maxHwUtil() or self.PaF_F_throughtput())) {
                    return false;
            }
            return true;
        }
        message : 'PaF <Operation> ' + self.name
        fix {
            title : "Move it to a new Component deployed to a new Node"
            do {
                self.moveToNewComponentDeployedOnNewNode();
            }
        }
    }
}
```

(a) Excerpt of an EVL pattern for PaF.



Blob <Artifact> BooksCatalog
Blob <Artifact> Database
Blob <Artifact> WebServer
CPS <Node> BooksControl Node
CPS <Node> BooksDispatcher Node
CPS <Node> Database Node
CPS <Node> MoviesControl Node
CPS <Node> MoviesDispatcher Node
CPS <Node> WebServer Node
PaF <Operation> getBookAvailability
PaF <Operation> login

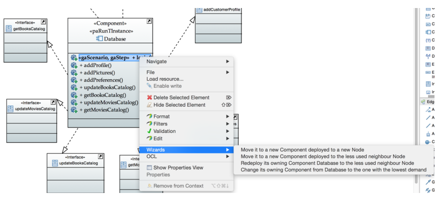
Move it to a new Component deployed to a new Node
Move it to a new Component deployed to the less used neighbour Node
Redeploy its owning Component Database to the less used neighbour Node
Change its owning Component from Database to the one with the lowest demand

(b) Refactoring session with EVL.

Fig. 5: EVL refactoring engine.

```
wizard PaF1 {
    guard :
        self.isTypeOf(Operation) and
        self.PaF_resDemand() and
        self.PaF_F_probExec() and
        (self.PaF_F_maxHwUtil() or self.PaF_F_throughtput())
    title : "Move it to a new Component deployed to a new Node"
    do {
        self.moveToNewComponentDeployedOnNewNode();
    }
}
```

(a) Excerpt of an EWL pattern for PaF.



Move it to a new Component deployed to a new Node
Move it to a new Component deployed to the less used neighbour Node
Redeploy its owning Component Database to the less used neighbour Node
Change its owning Component from Database to the one with the lowest demand

(b) Refactoring session with EWL.

Fig. 6: EWL refactoring engine.

Figure 7 graphically shows an EVL-based model refactoring. The upper side of the figure illustrates an excerpt of a performance-annotated software model, focusing on its Static and Deployment views, expressed as a UML Component and a Deployment Diagram, respectively. Under these diagrams, their tree-based textual representations within the Eclipse UML perspective are shown. After executing an antipattern detection step that uses the EVL engine of our approach, several actions are proposed to remove the detected antipatterns (see the popup in the middle of Figure 7). When the refactoring application is completed, a new model is obtained (see the bottom of the figure), where antipatterns have been removed. In the example of Figure 7, a detected occurrence of the Pipe and Filter antipattern is removed by applying the solution described in Section 2.1.

4.2 Sample Designer Support for the PaF Antipattern

The kind of support that our approach provides strictly depends on the execution semantics of the specific languages of the Epsilon ecosystem. The three languages that we consider in this paper allow to provide the following kind of refactoring sessions to the user.

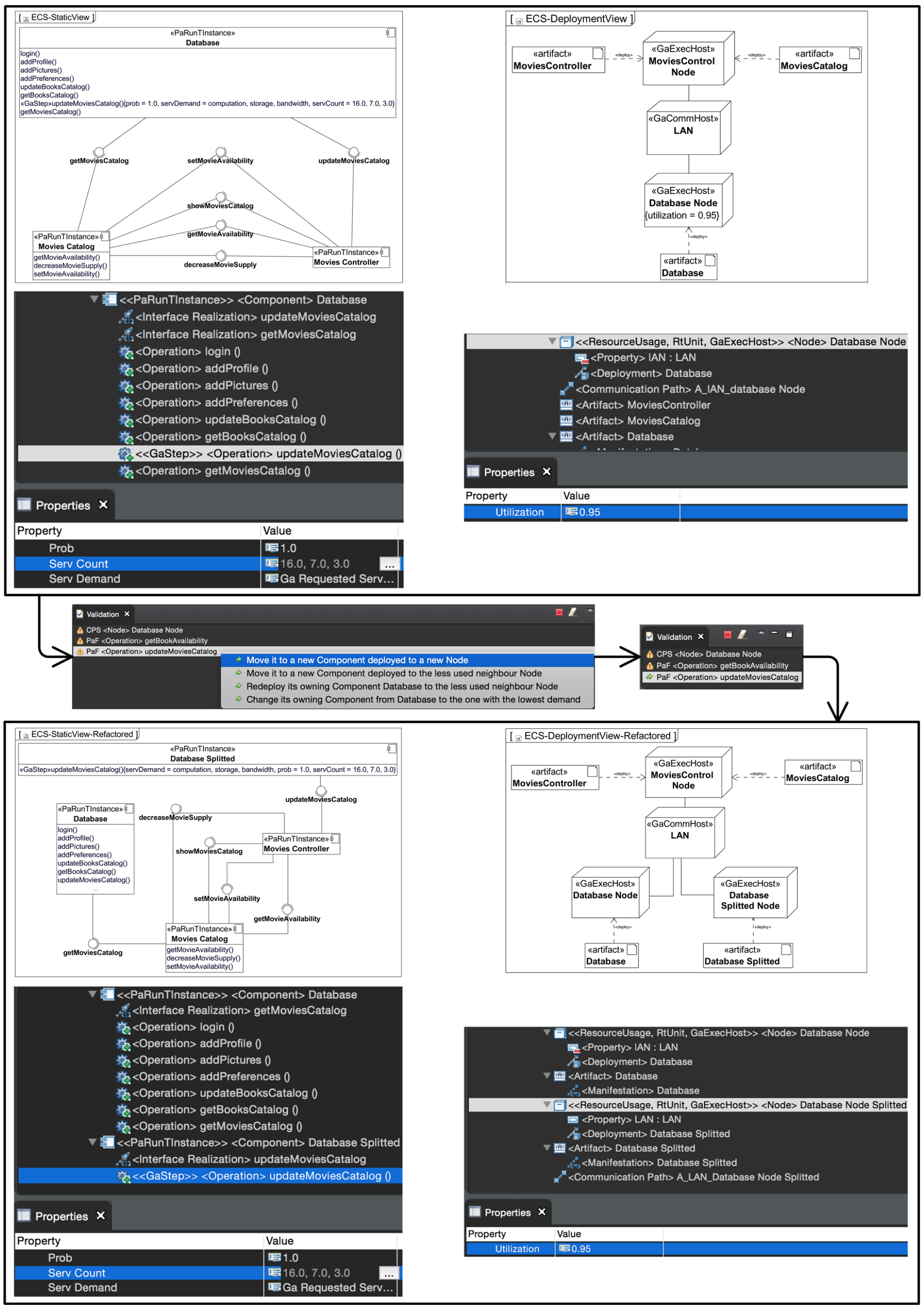
Batch refactoring sessions (see Figure 4b): implemented as EPL *pattern*s (see Figure 4a), they allow to execute the series of antipattern detection and refactoring specified in the corresponding .epl file. In case of iterative mode, the process is repeated until no more antipattern occurrences have been found or until the specified maximum number of iterations has been reached.

EPL provides the most limiting support to the designer, due to the fact that, in presence of two or more EPL patterns matching on identical conditions in the same .epl file, it is not possible to determine which pattern will be chosen by the Epsilon engine. For this reason, we assume that two EPL patterns matching on identical conditions cannot exist in the same .epl file. As a consequence, each antipattern can be codified only as one EPL pattern, thus limiting the solution to the unique randomly-chosen refactoring.

User-driven multiple refactoring sessions (see Figure 5b): implemented as EVL *critique*s (see Figure 5a), they allow to execute interactive antipattern detection and refactoring sessions, where antipattern occurrences are firstly detected on the software model, and a number of available refactorings (i.e. EVL *fix*es) are then selected by the user. Each selection immediately triggers the application of a refactoring on a temporary version of the software model. When the user stops the refactorng session, the temporary software model is finalized.

In the context of performance antipatterns detection and solution, we can assume that, even if more possible solutions are available for the same antipattern, they do not depend one each other (otherwise they could be collapsed into an unique refactoring). Moreover, since each refactoring is aimed at solving a detected antipattern occurrence, we can assume that the designer selects at most only one refactoring to apply for each detected occurrence.

User-driven single refactoring sessions (see Figure 6b): implemented as EWL wizards (see Figure 6a), they are directly integrated in the Eclipse-



<<PaRunTInstance>> <Component> Database
  <Interface Realization> updateMoviesCatalog
  <Interface Realization> getMoviesCatalog
  <Operation> login ()
  <Operation> addProfile ()
  <Operation> addPictures ()
  <Operation> addPreferences ()
  <Operation> updateBooksCatalog ()
  <Operation> getBooksCatalog ()
  <<GaStep>> <Operation> updateMoviesCatalog ()
  <Operation> getMoviesCatalog ()

Properties

Property	Value
Prob	1.0
Serv Count	16.0, 7.0, 3.0
Serv Demand	Ga Requested Serv…

<<ResourceUsage, RtUnit, GaExecHost>> <Node> Database Node
  <Property> lAN : LAN
  <Deployment> Database
<Communication Path> A_lAN_database Node
<Artifact> MoviesController
<Artifact> MoviesCatalog
<Artifact> Database

Properties

Property	Value
Utilization	0.95

Validation
CPS <Node> Database Node
PaF <Operation> getBookAvailability
PaF <Operation> updateMoviesCatalog
  Move it to a new Component deployed to a new Node
  Move it to a new Component deployed to the less used neighbour Node
  Redeploy its owning Component Database to the less used neighbour Node
  Change its owning Component from Database to the one with the lowest demand

Validation
CPS <Node> Database Node
PaF <Operation> getBookAvailability
PaF <Operation> updateMoviesCatalog

<<PaRunTInstance>> <Component> Database
  <Interface Realization> getMoviesCatalog
  <Operation> login ()
  <Operation> addProfile ()
  <Operation> addPictures ()
  <Operation> addPreferences ()
  <Operation> updateBooksCatalog ()
  <Operation> getBooksCatalog ()
  <Operation> getMoviesCatalog ()
<<PaRunTInstance>> <Component> Database Splitted
  <Interface Realization> updateMoviesCatalog
  <<GaStep>> <Operation> updateMoviesCatalog ()

Properties

Property	Value
Prob	1.0
Serv Count	16.0, 7.0, 3.0
Serv Demand	Ga Requested Serv…

<<ResourceUsage, RtUnit, GaExecHost>> <Node> Database Node
  <Property> lAN : LAN
  <Deployment> Database
<Artifact> Database
  <Manifestation> Database
<<ResourceUsage, RtUnit, GaExecHost>> <Node> Database Node Splitted
  <Property> LAN : LAN
  <Deployment> Database Splitted
<Artifact> Database Splitted
  <Manifestation> Database Splitted
<Communication Path> A_LAN_Database Node Splitted

Properties

Property	Value
Utilization	0.95

Fig. 7: Example of model refactoring effect in Epsilon.

based modeling environment of the user (e.g., Papyrus[2]), thus automatically detecting performance antipattern occurrences whenever an element is selected in the modeling environment. They also enable solutions among which the user can select the one to apply immediately on the software model, thus carrying out a refactored one. At this point, the refactoring session is terminated, and a new one might start.

Differently from EPL patterns, two EWL wizards matching on identical conditions can exist in the corresponding .ewl file. As a result, similarly to EVL, more than one solution can be enabled with EWL.

5 Conclusion and Future Work

In this paper, we have proposed a first step towards an approach that embeds performance antipatterns detection and solution within an unique supporting environment. In the context of the Epsilon platform, which provides an ecosystem of task-specific languages, interpreters and tools for MDE activities, we have selected EPL, EVL, and EWL as basis for three engines that provide different kinds of support to performance-driven software model refactoring.

Several aspects have been discussed in this paper and some other ones have been just mentioned. However, we need to work on them in the near future.

So far, we have implemented occurring conditions and refactorings for four performance antipatterns over twelve existing ones. Hence, we aim at extending our work to other performance antipatterns. Beside this, two next steps ahead in process automation, aimed at reducing redundant coding effort, shall be: (i) generating code from antipattern formulations, and (ii) porting the code among the three considered languages.

Finally, for sake of a future vision, we like to mention two long-term goals.

Metamodel-independence. In general, our approach is metamodel-independent, in the sense that its founding idea works for any modeling notation. However, we have experimented it on UML software models profiled with the MARTE profile. Nevertheless, several other performance-oriented modeling languages exist, e.g., Palladio and Æmilia. With this respect, we intend to enlarge the spectrum of modeling notations addressed by our approach. The approach shall be based on a pivot language (i.e., a DSL transparent to the final user) representing a neutral notation, whose constructs can be mapped to the ones of other modeling languages.

Threshold values. As discussed in [11,12], thresholds cannot be avoided in the performance antipatterns definition, and their multiplicity and estimation accuracy heavily influence both detection and refactoring activities. With this respect, we aim at extending our approach by introducing heuristics for assigning values to antipattern thresholds. Such heuristics would also assign probability and effectiveness values to detected antipattern occurrences and available refactoring actions, respectively, as discussed in [17].

[2] `https://www.eclipse.org/papyrus/`

References

1. C. Smith, “Introduction to software performance engineering: Origins and outstanding problems,” in *Formal Methods for Performance Evaluation*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4486, pp. 395–428.
2. H. Koziolek, “Performance evaluation of component-based software systems: A survey,” *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, Aug. 2010.
3. D. Arcelli and V. Cortellessa, “Assisting Software Designers to Identify and Solve Performance Problems,” in *First International Workshop on the Future of Software Architecture Design Assistants (FoSADA), WICSA and CompArch 2015*, Montréal, Canada, CA, May 2015.
4. D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez, “The epsilon book,” *Structure*, vol. 178, pp. 1–10, 2010.
5. W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., 1998.
6. C. U. Smith and L. G. Williams, “Software performance antipatterns,” in *Proc. of the 2nd ACM International Workshop on Software and Performance.* ACM, 2000.
7. ——, “Software Performance AntiPatterns; Common Performance Problems and their Solutions,” in *CMG Conference*, 2001, pp. 797–806.
8. ——, “New software performance antipatterns: More ways to shoot yourself in the foot,” in *CMG Conference*, 2002, pp. 667–674.
9. ——, “More new software antipatterns: Even more ways to shoot yourself in the foot,” in *CMG Conference*, 2003, pp. 717–725.
10. V. Cortellessa, A. Di Marco, and C. Trubiani, “An approach for modeling and detecting software performance antipatterns based on first-order logics,” *Software and Systems Modeling*, vol. 13, no. 1, pp. 391–432, 2014.
11. D. Arcelli, V. Cortellessa, and C. Trubiani, “Influence of numerical thresholds on model-based detection and refactoring of performance antipatterns,” *First Workshop on Patterns Promotion and Anti-patterns Prevention*, 2013.
12. ——, “Experimenting the influence of numerical thresholds on model-based detection and refactoring of performance antipatterns,” *ECEASST*, vol. 59, 2013.
13. D. Tamzalit, B. Schätz, A. Pierantonio, and D. Deridder, “Introduction to the sosym theme issue on models and evolution,” *Software and System Modeling*, vol. 13, no. 2, pp. 621–623, 2014.
14. A. Diaz-Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann, “Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant,” in *Quality of Software Architectures, Models and Architectures*, ser. LNCS, 2008, vol. 5281, pp. 171–188.
15. N. Mani, D. C. Petriu, and C. M. Woodside, “Studying the Impact of Design Patterns on the Performance Analysis of Service Oriented Architecture,” in *EUROMICRO Conference on Software Engineering and Advanced Applications*, 2011, pp. 12–19.
16. T. Arendt and G. Taentzer, “Integration of smells and refactorings within the eclipse modeling framework,” in *Fifth Workshop on Refactoring Tools 2012, WRT ’12*, 2012, pp. 8–15.
17. D. Arcelli, V. Cortellessa, and C. Trubiani, “Performance-based software model refactoring in fuzzy contexts,” in *Fundamental Approaches to Software Engineering*, 2015, pp. 149–164.