Software Model Refactoring Driven by Performance Antipattern Detection

Vittorio Cortellessa
University of L'Aquila, Italy
vittorio.cortellessa@univaq.it

Daniele Di Pompeo
University of L'Aquila, Italy
daniele.dipompeo@univaq.it

Vincenzo Stoico
University of L'Aquila, Italy
vincenzo.stoico@graduate.univaq.it

Michele Tucci
Charles University, Czech Republic
tucci@d3s.mff.cuni.cz

ABSTRACT

The satisfaction of ever more stringent performance requirements is one of the main reasons for software evolution. However, determining the primary causes of performance degradation is generally challenging, as they may depend on the joint combination of multiple factors (e.g., workload, software deployment, hardware utilization). With the increasing complexity of software systems, classical bottleneck analysis seems to show limitations in capturing complex performance problems. Hence, in the last decade, the detection of performance antipatterns has gained momentum as an effective way to identify performance degradation causes. In this tool paper we introduce PADRE (Performance Antipattern Detection and REfactoring), a tool for: (i) detecting performance antipattern in UML models, and (ii) refactoring models with the aim of removing the detected antipatterns. PADRE has been implemented within Epsilon, which is an open-source platform for model-driven engineering, and it grounds on a methodology that allows performance antipattern detection and refactoring within the same implementation context.

1. INTRODUCTION

The complexity and heterogeneity of software systems makes the performance requirements validation difficult to be carried out. A performance requirement can impose a (strict) limit on a combination of attributes by different nature. For example, a performance requirement might affect the software deployment and/or the largest workload affordable for a system.

Traditionally, performance analysis has been based on bottleneck identification and removal. However, the bottleneck analysis might be not enough to tackle the complexity of modern systems. For this reason, performance antipatterns have recently emerged as more effective instruments to detect and remove non-trivial performance problems. A performance antipattern is a formalization of a bad design practice that might induce performance degradation [24]. Furthermore, performance antipatterns can describe performance flaws both on models [7] and on code [23].

In this paper, we present PADRE (Performance Antipattern Detection and REfactoring), a tool for the detection and removal of performance antipatterns in UML models. PADRE is developed on the basis of a well-assessed approach for the formalization of performance antipattern conditions with first-order logic [13], and on a methodology for model refactoring aimed at removing performance antipatterns [4, 5]. It has been designed and implemented within Epsilon [20], which is a widely adopted model-driven engineering platform.

The novelty of PADRE is that it introduces a high degree of automation in performance antipatterns detection and removal while keeping the human contribution in the loop. Therefore, one can produce modeling alternatives that might show better performance. In perspective, it can be adopted as a performance evaluation and improvement tool even in production environments by connecting monitoring data gathered from a running system to its corresponding model elements. Those model elements could be fed with actual performance measurements coming from profiling through MARTE [1] stereotypes. Once a possible beneficial evolution is identified, the corresponding refactoring could be (semi-)automatically propagated to the implementation level [6, 15].

The rest of this paper is organized as follows: Section 2 discussed related work; Section 3 introduces PADRE goals, requirements, architecture and inner behavior; Section 4 shows PADRE at work by means of an illustrative example.

2. RELATED WORK

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. In practice, the application of refactoring actions on models helps to identify the best refactoring paths before expensively applying them on the code itself.

Most of the recent papers in this area deal with model-based refactoring driven by functional properties [8, 18, 19, 22], whereas only few of them consider non-functional prop-

erties [14, 21, 17].

Cortellessa [11] reports on several approaches introduced in the literature to use performance antipattern knowledge for detecting and removing performance problems in software models.

Xu [25] 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. However, as it has been shown in [3], 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.

Recent studies [16, 14] place the antipattern detection and removal within multi-objective optimization problems. They show that the performance antipattern detection can improve the quality of Pareto frontier in terms of performance although different modelling notations (*e.g.,* UML, and Æmilia) were used.

At the best of our knowledge, PADRE is the first integrated tool that implements, within a model-driven environment (*i.e.,* Epsilon), an approach for antipattern detection and removal on UML models.

3. PADRE TOOL

In this section, we describe goals, requirements, the architecture, and the underlying methodology of PADRE.

3.1 Goals

PADRE allows model refactoring driven by performance antipatterns detection. PADRE supports UML models profiled with MARTE as architectural description language, as well as Queueing Networks (QN) and Layered Queueing Networks (LQN) as performance model notations. The MARTE profile extends the UML language with performance concerns. In particular, it allows setting performance input parameters as well as reporting performance analysis results back to the UML model.

A key goal of PADRE is providing support to model refactoring driven by performance antipatterns detection in a single integrated environment, which helps designers and performance experts to work together. In Addition, PADRE exploits the Epsilon platform [20] to perform model-driven tasks.

3.2 Tool requirements

In order to allow performance antipatterns detection and removal, PADRE works on UML models made up of three views of the system as depicted in Figure 1:

- *Static view* is represented through a UML Component Diagram. In this view, we use UML Components and UML Operations. Furthermore, components interface realizations and interface usages describe relationship among UML Components.

- *Deployment view* is represented through a UML Deployment Diagram, which describes manifestation of UML Components through UML Artifacts, UML Devices (i.e. platform nodes), and the deployment relationship between UML Devices and UML Artifacts.

- *Dynamic view* is represented through a set of UML Sequence Diagrams, one for each UML Use Case, where UML Components are associated to UML Lifelines while UML Message signatures represent UML Operation calls provided by recipient UML Lifeline.

PADRE considers only stereotypes belonging to the Generic Quantitative Analysis Modeling (GQAM) package of MARTE, as summarized in the following, and Figure 1 depicts their usage:

- The GaExecHost stereotype is applied to UML Device instances. It defines the scheduling policy (*schedPolicy*), the device speed (*speedFactor*), and its multiplicity (*resMult*). In addition, the tagged value *utilization* contains the utilization result of a performance analysis.

- The GaStep and GaWorkloadEvent stereotypes are applied to UML Use Cases. GaStep defines the execution probability for the Use Case (*prob*), the number of repetitions (*rep*), whereas GaWorkloadEvent defines the workload arrival pattern (*pattern*).

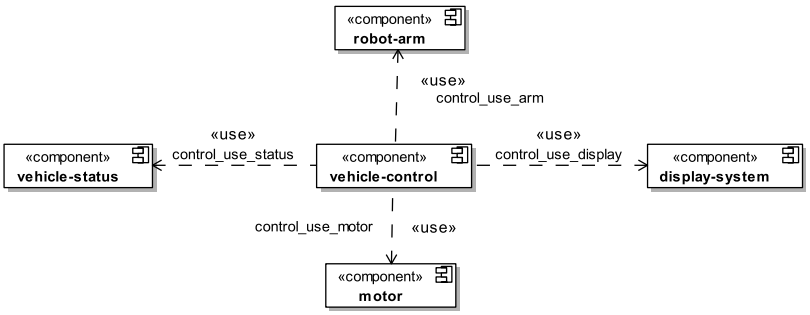
- The GaScenario is applied to UML Use Cases to report performance analysis results. In particular, the *respT*, and the *throughput* tagged values contain response time and throughput values obtained by a performance analysis.

- The GaAcqStep stereotype is applied to UML Messages to specify the execution probability (*prob*), the number of repetitions (*rep*), the payload (*msgSize*), and the service demand (*servDemand*) for the operation request.

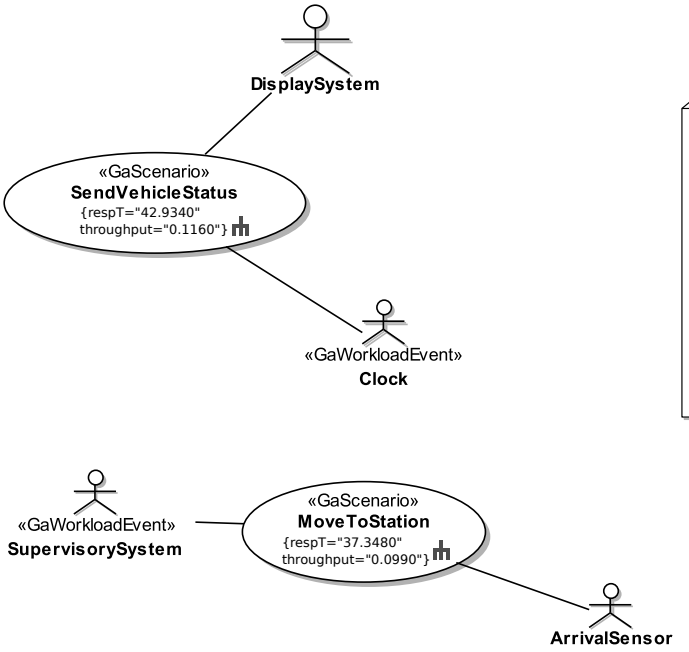
PADRE allows the selection of two performance model notations, which are Queueing Networks and Layered Queueing Networks. The user decides the target notation, being aware that PADRE uses the MVA [9] implementation provided by the JMT toolset (*i.e.,* JMVA) [10] for solving QN models, whereas the native provided solver for LQN models.

3.3 Tool architecture

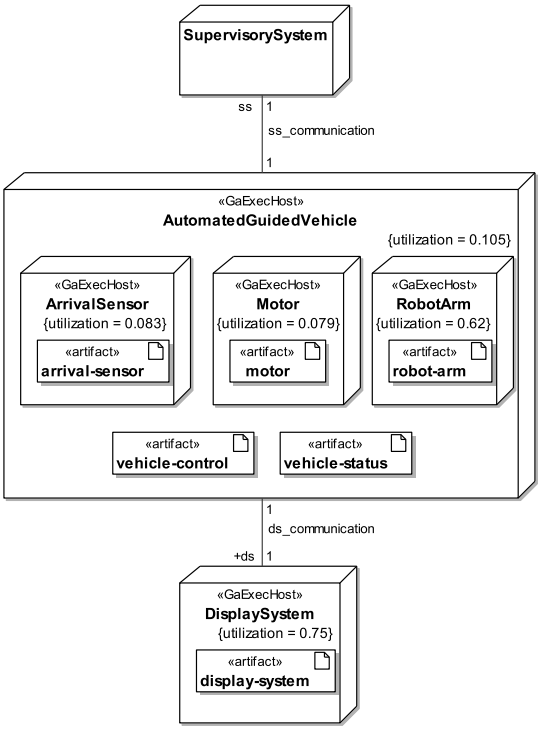
Figure 2 illustrates the architecture of PADRE. We have introduced the *PadreValidationView*, which extends the *ValidationView* of the Epsilon Validation View (see *evl* component). In particular, our view allows the user to choose from a variety of refactoring options. Each refactoring action triggers the process that applies the action and creates a new refactored model. The *refactoring* component, instead, includes the engine and the refactoring actions. The *transformation* component is responsible for automating model-to-model transformations. This component converts the system model (*i.e.,* UML) to a performance model (*e.g.,* LQN). Then, the specific solver executes the performance model and provides performance indices. All the third-party elements related to the performance model are represented in Figure 2 by the *Performance Model Resources* artifact. The ModelFactory class takes advantage of these resources to build an internal representation of the performance model.



(a) Static View.



(b) Use Case.



(c) Deployment View.

Figure 1: A UML-MARTE example model.

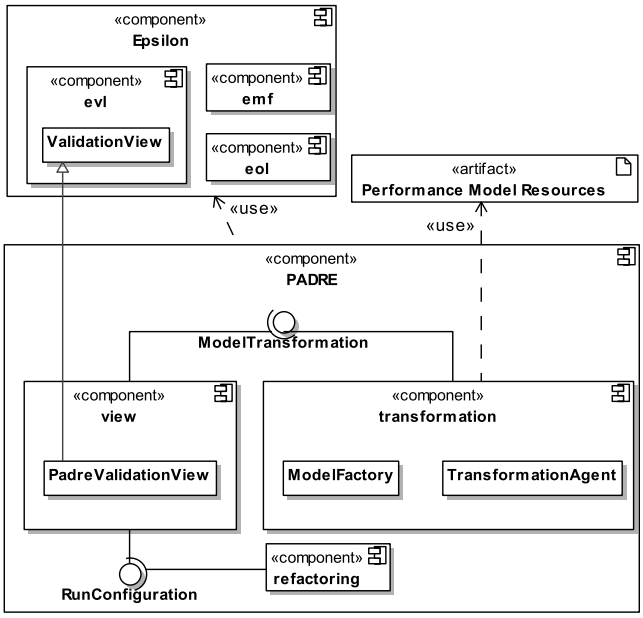


Figure 2: PADRE architecture.

The TransformationAgent exploits this internal representation to execute a model-to-model transformation. The transformation uses the system model as input and the performance model as output. Therefore, the TransformationAgent triggers the solver to obtain the performance estimation. The separation between model creation and transformation management facilitates the integration of several performance models into PADRE.

3.4 Tool workflow

Figure 3 describes the iterative methodology underlying PADRE.

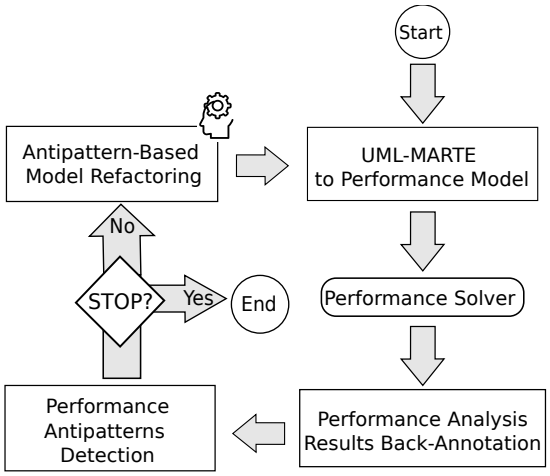


Figure 3: PADRE workflow.

The process starts with the execution of a *UML-MARTE-to-Performance-model* transformation, which consumes the system model and produces the corresponding performance model (*e.g.,* MVA, or LQN). The produced performance model will be solved (or simulated) by the `Performance Solver`. Thus, the obtained performance indices are fed back to the system model through the *Performance Analysis Results Back-Annotation* activity.

Once the system model has been annotated, the *Performance Antipatterns Detection* activity takes place, followed by user-driven *Antipattern-Based Model Refactoring* activity. Such activities represent key features of the PADRE methodology in proving refactoring solutions to meet the performance requirements.

The *Performance Antipattern Detection* and the subsequent *Antipattern-Based Model Refactoring* activities have been implemented in EVL and EOL modules. In particular, the detection module implements antipattern rules as EVL constraints that will be verified on the system model, as depicted in Figure 4a. Currently, PADRE can detect seven performance antipatterns described in [12]. In detail, we implemented the first order logic of each performance antipattern as an EVL constraint (see the *check* block in Figure 4a), and we associated, to each antipattern, possible refactoring actions that have shown the ability to remove it (see a *fix* block in Figure 4a).[1] The refactoring module is made up of EOL functions that are able to change the system model (see the operation invoked in the *do* block in Figure 4a).

PADRE provides a unifying working environment, by exploiting the Epsilon platform, that allows performance analysis interpretation and feedback generation. The feedback generation is one of the major contribution of PADRE. In particular, PADRE generates performance feedback through checking properties on the system model. On the basis of this feedback, PADRE enables a user-driven selection of refactoring actions. The refactoring actions are aimed at satisfying performance properties, such as performance requirements. PADRE performance antipattern portfolio, which is made up of detection rules and refactoring actions, is described through an EVL module.

The refactoring actions available in the current portfolio have been designed to obtain refactored models that can still be solved. Although we have not formally proven this aspect, our experimental testing has not found a counterexample up to now. The process ends either arbitrarily by stopping the workflow execution, or once the performance requirements are met.

3.5 Target and application

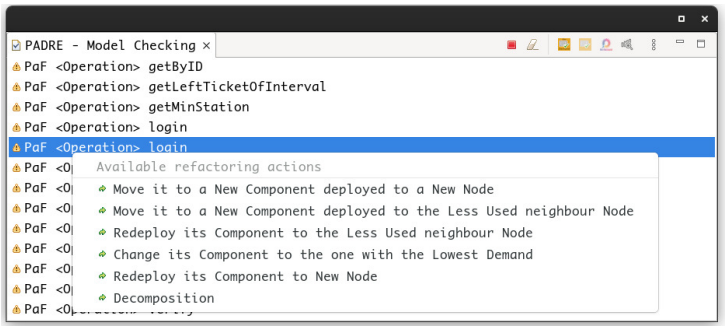
PADRE has been designed and implemented to be employed in real scenarios and for research scopes. In the context of a real scenario, performance antipattern detection and removal should help designers and performance experts to identify system architecture alternatives that show better performance than the initial architecture. Furthermore, the benefits of using PADRE in a real scenario are twofold: reducing the number of manually generated architecture alternatives, obtaining architecture alternatives that should not be affected by bad design practices that might lead to future performance degradation.

Regarding the context of research activities, PADRE opens a new window on the performance antipattern detection and removal research area. In particular, it allows one to explore different refactoring actions, which might discover even better architecture alternatives. On the other hand, PADRE

[1]PADRE - `https://github.com/SEALABQualityGroup/padre/tree/tosme-tool-demo`.

```
context Operation {
    critique PaF {
        check:
            not (self.PaF_F_probExec() and self.PaF_F_resDemand() and
                 (self.PaF_F_maxHwUtil() or self.PaF_F_throughput()))
        message : "PaF <Operation> " + self.name
        fix {
            title : "Move it to a new Component deployed to a new Node"
            do {
                self.moveToNewComponentOnNewDevice();
            }
        }
        fix {
            title : "Move it to a new Component deployed to the" +
                    " less used neighbour Node"
            do{
                self.moveToNewComponentOnLessUsedNearDevice();
            }
        }
    }
}
```

(a) Excerpt of a rule for the Pipe and Filter antipattern.



(b) Example of PADRE refactoring session.

Figure 4: Performance antipatterns detection and model refactoring with PADRE

has been designed to be extensible as much as possible, and it can be applied to different modelling notation on which few research studies have been conducted until now.

The ability of PADRE in the context of performance analysis has been shown within the European Project Megamart2 in which practitioners and researchers worked together in the context of megamodelling [2].

4. ILLUSTRATIVE EXAMPLE

In this section, we show PADRE at work by means of a running example.

Figure 4 (a) shows an excerpt of *Pipe and Filter* (PaF) performance antipattern definition [13], *i.e.,* the detection rule and some of the available refactoring action(s). A performance antipattern definition is an EVL `critique` that applies to a certain context (i.e. a UML metaclass) and that contains an antipattern detection rule (namely `check`s), a message to return when the antipattern is detected and a set of imperative blocks (namely `fix`es), each codifying a possible model refactoring.

For example, the PaF detection rule is applied to UML Operations, and it consists of four EVL operations returning a boolean (namely *PaF_F_probExec()*, *PaF_F_resDemand()*, *PaF_F_throughtput()*, and *PaF_F_maxHwUtil()*), each representing a predicate of a first-order logic formula in conjunctive normal form.

The `do` block of a `fix` contains a call to an ad-hoc EVL operation (see `moveToNewComponentOnNewDevice` and `moveToNewComponentOnLessUsedNearDevice` of Figure 4 (a)), which codifies a possible model refactoring that might lead to the removal of the PaF occurrence and, hopefully, to a performance improvement.

The kind of support currently provided by PADRE is named *User-driven multiple refactoring* [4], and it strictly depends on the execution semantics of the EVL language [20]. In particular, it consists of interactive antipattern detection and refactoring sessions, where antipattern occurrences (*i.e., critiques*) are firstly detected on the software model, and a number of available refactorings (*i.e., fixes*) are then selected by the user. Each selection immediately triggers the application of the refactoring on a temporary version of the model. When the user stops the refactoring session, the temporary software model is finalized. Note that such "freezing" of the refactoring session, which is native for the EVL execution engine, is not suitable in our context, because new performance indices are needed for a new antipattern detection. Moreover, if a new element is created by a refactoring action, then the latter cannot be referred in subsequent refactoring. For this reason, PadreValidationView (see Figure 2) overrides such native EVL engine behavior by updating each temporary refactored version of the model with performance indices obtained from the performance analysis.

Figure 4 (b) shows a snapshot of PadreValidationView right after a performance antipatterns detection: all the detected occurrences are listed in the validation view and, for each occurrence, both the context metaclass and the metaclass instance representing the antipattern source are listed. By right-clicking a list item, PADRE proposes available refactoring actions aimed at removing such antipattern occurrence, and the user can arbitrarily apply one of them. For example, among the 11 occurrences (*i.e.,* 3 Blobs, 6 CPSs and 2 PaFs) [13] reported in Figure 4 (b), four refactoring actions are applicable to the PaF having the *login* Operation as source: the first two, namely *Move it to a new Component deployed to a new Node* and *Move it to a new Component deployed to the less used neighbour Node*, correspond to the `fix`es of Figure 4 (a) and, in particular, to `moveToNewComponentOnNewDevice` and `moveToNewComponentOnLessUsedNearDevice` EVL operations, respectively.

5. REFERENCES

[1] A UML profile for MARTE: modeling and analysis of real-time embedded systems. OMG, 2008.

[2] W. Afzal, H. Brunelière, D. D. Ruscio, A. Sadovykh, S. Mazzini, E. Cariou, D. Truscan, J. Cabot, A. Gómez, J. Gorroñogoitia, L. Pomante, and P. Smrz. The megam@rt2 ECSEL project: Megamodelling at runtime - scalable model-based framework for continuous development and runtime validation of complex systems. *Microprocess. Microsystems*, 61:86–95, 2018.

[3] D. Arcelli and V. Cortellessa. Software model refactoring based on performance analysis: better working on software or performance side? In *FESCA'13*, pages 33–47, Feb. 2013.

[4] D. Arcelli, V. Cortellessa, and D. Di Pompeo. Performance-driven software model refactoring. *Information & Software Technology*, 95:366–397, 2018.

[5] D. Arcelli, V. Cortellessa, and D. Di Pompeo. Automating performance antipattern detection and software refactoring in UML models. In X. Wang,

D. Lo, and E. Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 639–643. IEEE, 2019.

[6] D. Arcelli, V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci. Exploiting architecture/runtime model-driven traceability for performance improvement. In *IEEE International Conference on Software Architecture*, pages 81–90, 2019.

[7] D. Arcelli and D. Di Pompeo. Applying design patterns to remove software performance antipatterns: A preliminary approach. In E. M. Shakshuki, editor, *The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017)*, volume 109 of *Procedia Computer Science*, pages 521–528, 2017.

[8] J. Bansiya and C. G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[9] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, Apr. 1975.

[10] M. Bertoli, G. Casale, and G. Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[11] V. Cortellessa. Performance Antipatterns: State-of-Art and Future Perspectives. In *EPEW'13*, pages 1–6. Springer, 2013.

[12] V. Cortellessa, A. Di Marco, and C. Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *SoSyM*, 13(1):391–432, 2014.

[13] V. Cortellessa, A. Di Marco, and C. Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *Software and System Modeling*, 13(1):391–432, 2014.

[14] V. Cortellessa and D. Di Pompeo. Analyzing the sensitivity of multi-objective software architecture refactoring to configuration characteristics. *Inf. Softw. Technol.*, 135:106568, 2021.

[15] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci. A model-driven approach for continuous performance engineering in microservice-based systems. *Journal of Systems and Software*, 183:111084, 2022.

[16] V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci. On the impact of performance antipatterns in multi-objective software model refactoring optimization. In M. T. Baldassarre, G. Scanniello, and A. Skavhaug, editors, *47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021, Palermo, Italy, September 1-3, 2021*, pages 224–233. IEEE, 2021.

[17] V. Cortellessa, R. Eramo, and M. Tucci. From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement. *Inf. Softw. Technol.*, 127:106362, 2020.

[18] R. Fourati, N. Bouassida, and H. Ben Abdallah. A Metric-Based Approach for Anti-pattern Detection in UML Designs. *Computer and Information Science*, 364(Chapter 2):17–33, 2011.

[19] L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, and D. Varró. Using Graph Transformation for Practical Model-Driven Software Engineering. In *Model-Driven Software Development*, pages 91–117. Springer Berlin Heidelberg, 2005.

[20] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez. *The epsilon book*. Structure, 2010.

[21] A. Koziolek, H. Koziolek, and R. Reussner. PerOpteryx: automated application of tactics in multi-objective software architecture optimization. In *QoSA'11*, pages 33–42. ACM, 2011.

[22] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *SoSyM*, 6(3):269–285, 2007.

[23] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–91, 2008.

[24] C. U. Smith and L. G. Williams. Software performance antipatterns for identifying and correcting performance problems. In *CMGC'12*, 2012.

[25] J. Xu. Rule-based automatic software performance diagnosis and improvement. *Perf. Eval. Journal*, 69(11):525–550, Nov. 2012.