Model-based software refactoring driven by performance analysis

Davide Arcelli

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi di L'Aquila
67100 L'Aquila, Italy
{davide.arcelli}@univaq.it

Abstract. In order to deal with performance of software systems, it is important to introduce approaches that work in the early phases of the software life-cycle, even before the code is developed. In fact, if performance requirements are not met, there may be negative consequences on significant parts of the project.
Some work has been done in the last few years to tackle the problem of automatically interpreting model-based performance analysis results and translating them into architectural feedback. In this context, software and performance models are typical artifacts involved in the interpretation, and architectural feedback consists of refactoring that can take place either on the software or the performance model. We present here this problem context, by means of a unifying framework that supports model-based software refactoring driven by performance analysis.

Keywords: Software Performance Engineering, Performance Antipatterns, Software Refactoring, Model-Driven Engineering, Control Theory.

1 Problem

In the software development domain, there exists a high interest in the early validation of performance requirements because it avoids late and expensive fix to consolidated software artifacts [1]. Current approaches to these problems are mostly based on the skills and experience of software developers or performance analysts. Since manual tasks are cost-intensive and are error-prone due to the complex design space, it is crucial to introduce automation in this domain [2]. One main challenge is to enable software designers and/or performance experts (that represent the target audience of this this research) to automatically analyze solutions to performance problems. In other words, we focus on sources of performance problems, both potential and existing, to suggest how to refactor models in order to remove them. For example, it may happen that an excessive number of requests is required to perform a task, due to inefficient use of available bandwidth, an inefficient interface, or both. To mitigate this problem, the model can be refactored by automatically introducing the Batching performance pattern for a better use of available bandwidth [3], or the Session Facade design pattern to provide more efficient interfaces [4].

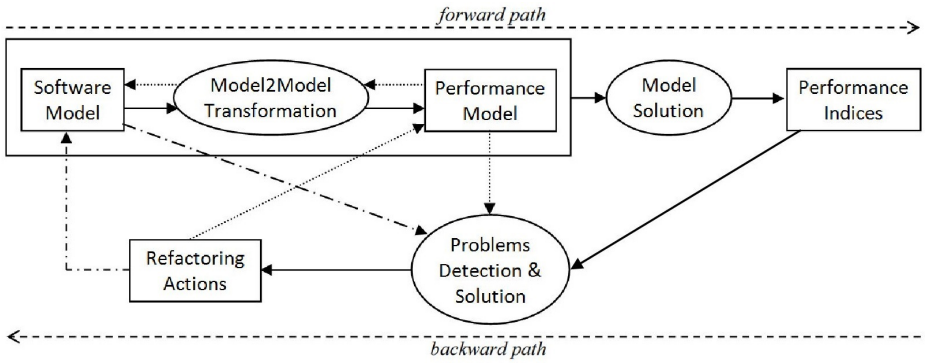

Fig. 1: Software performance analysis process.

Figure 1 illustrates a model-based performance analysis round-trip process that represents the context of this work. Continuous arrows represent the process control flow.

The forward path of the process starts from a software model that is transformed into a performance model [5] that can be solved with common analysis techniques/tools to obtain performance indices (e.g., response times and throughputs, node utilizations, node queue lengths) [6,7]. The backward path consists of a main step aimed at detecting and removing possible sources of performance problems. A set of refactoring actions is produced that may apply to the software or the performance model.

Due to the target audience of this research, consisting of software designers and performance experts, we identify two "sides" (i.e., categories) of related work, based on the artifacts with which these actors cope with. In particular, (i) at the *software side* (see dotted-line arrows), software designers work on a software model expressed in a certain modeling notation, e.g., UML profiled with MARTE [8], whereas (ii) at the *performance side* (see dotted arrows), performance experts work on a performance model, e.g., a Queueing Network (QN) [5]. The choice of the side to apply the refactoring is crucial, because software and performance models differ both syntactically and semantically [9]. In case of performance side, a transformation from the performance model to the software model may have to take place when satisfactory performance indices are obtained [10]. However, in both cases the forward path has to be run at each iteration to obtain the performance indices of a refactored (performance or software) model.

Each step of the framework can be implemented with different technologies and methods. The main objective of this work is to propose (automated) solutions to the steps of the framework, by leveraging implementations of the process that use technologies and methods from *Model-Driven Engineering* (MDE) and *Control Theory*.

2 Related and Prior works

An extensive overview of existing research in the field of software refactoring (not only related to performance problems) is provided in [11].

In literature, many approaches often apply refactoring to the program itself (i.e. the source code), but preventing performance problems at design-time is clearly suitable to avoid critical situations that lead to dramatically raise development costs [12].

Some model-based approaches for automated performance diagnosis and improvement have been introduced up today in the software modeling domain [13,14,15,16], but they strictly depend on specific modeling notations.

In the last two decades the concept of *Performance Antipattern* [17] has been used for "codifying" the knowledge and experience of analysts, by looking at negative features of a software system [18,19]. A performance antipattern identifies a *problem*, i.e. a bad practice that negatively affects the software performance, and a *solution*, i.e. a set of refactoring actions that can be carried out to remove it.

We have been working, in the last few years, to provide tailored techniques to manage technology-independent performance antipatterns on software models. However, still a lot of work is ahead of us, as illustrated in the rest of the paper, due to their intrinsic complexity (as combinations of design features and performance indices) and their multi-view nature (as combinations of elements from different modeling views).

In the last few years, *Control Theory* [20] has started to be applied to analytically guarantee *Quality of Service* in unpredictable environments [21,22,23]. In general terms, Control Theory is a systematic approach whereby a signal to be controlled is compared to a desired reference signal, and the discrepancy is used to compute corrective control actions, thereby making the system able to adapt to specific contexts, even in presence of disturbances.

On one end, an important contribution concerning the application of Control Theory to performance analysis came from Lu [24], where queueing systems have been preliminarily considered to be controlled. On another end, a whole theory on bottleneck identification and removal on *Queueing Networks* has been introduced by Lazowska [5] few decades ago, and has been continuously refined by more recent results [25,26] that propose approaches for automated software performance diagnosis by identifying performance flaws before the software system implementation. Thus, we have started to study the problem of automated refactoring at the performance side as solvable with the (formal) synthesis of controllers on performance models, aimed at keeping the indices of the latter (e.g., node queue lengths) within pre-defined ranges, also in presence of significant divergences of variable parameters (e.g., workload, operational profile).

3 Proposed Solution

So far, we have deeply explored solutions at the software side, hence Section 4 focuses on the latter. At the performance side, we have clear in mind our contribution and how to obtain it, but we are in a preliminary phase of the work.

On the basis of this recent experience and the previous work done by our research group on detecting and solving performance antipatterns in different modeling languages, i.e., UML profiled with MARTE [13], PCM [14], and Æmilia [16], we make an abstraction step to define a metamodel for specifying performance antipatterns and refactoring actions, conformingly to the logic-based definitions given in [27] and independently from users modeling notations. The metamodel we propose is named *Performance Antipatterns Modeling Language* (PAML). It specifies concepts on which the constructs of modeling languages used by software designers can be mapped. Such

mapping enables the porting of advanced MDE techniques (e.g., model differences for antipattern solution [28]) on concrete modeling languages.

Primary concepts of the performance antipatterns domain come from the union of constructs in two "categories": (i) *Reference Elements for Antipatterns Definition* (READ), whose constructs are mapped onto the design constructs of the considered modeling languages, and (ii) *performance elements*, whose constructs are mapped onto the performance constructs of the considered modeling languages. For example, READ contains the concept of *SoftwareEntity*, aimed at specifying a software resource that owns *Operation*s. A performance element named *StructuredResourceDemand* is used to specify the amount of service that an operation requires to a set of sw/hw resources.

READ is a *Role-Based Modeling Language* (RBML), i.e., a language for describing role models. The concept of role is very suitable to capture the heterogeneity of the knowledge (i.e. design features and performance indices) underlying the specification of performance antipatterns. We embed READ into PAML while driven by clear separation of concerns (e.g., we explicitly characterize parts of the system in terms of *Static*, *Dynamic* and *Deployment View*s). This key-feature of PAML is preserved in antipattern solutions, where the concept of *role model* [29] explicitly emerges to model refactoring actions aimed at falsifying antipattern logical predicates, in terms of the model difference between a *source role model* (SRM) and the corresponding *target role model* (TRM) [28], still relying on READ constructs.

At the performance side, we apply feedback control theory to performance models representing adaptive software, in order to ensure that performance requirements are met. We aim at defining a broadly applicable methodology for the design of control systems based on QN models with formally provable quality guarantees. To do this, we define *Adaptive Queueing Networks* (AQNs), i.e., QNs where controllers can be synthesized and embedded to support adaptation at simulation time. We envisage a service center as a plant, where a performance index subject to a quantified requirement (e.g., response time, queue length) can be sensed by a controller; modifiable model parameters represent variables that can be manipulated by the controller through actuators (e.g., demands of classes of jobs to service centers); disturbances can come from the system environment (e.g., variable workload and/or operational profile over time). The goal of a controller in this context is to keep the sensed value within a certain threshold (setpoint) by acting on model parameters under disturbances.

4 Preliminary work

First, we have conducted a study aimed at highlighting the peculiar aspects of working on either the software side or the performance side. Here we shortly report some important findings related to modeling notations and process convergence, i.e., MN and PC respectively. Results of the study are explicitly discussed in [9].

MN_1: The number of alternative refactoring actions available at the software side is usually considerably higher than the one at the performance side.

MN_2: The refactoring complexity is generally lower at the performance side.

PC_1: The effectiveness of refactoring actions will be given by the trade-off between the refactoring complexity (i.e. the distance between the original model and the refactored one) and the performance gain obtained from the refactoring.

PC_2: Although the number of refactoring actions is typically higher at the software side, the performance gain deriving from refactoring actions is more unpredictable. For this reason, decision support heuristics, for example based on convenience metrics, might help the designer to mitigate this aspect [30,2] at the software side.

PC_3: Since the refactoring complexity is generally lower at the performance side, more iterations may be needed to converge at this side. Nevertheless, formal support and high degree of automation can be more easily achieved at the performance side.

In [27] a formal interpretation is provided, based on first-order logic rules, that defines a set of system properties under which a performance antipattern occurs on 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). Both the detection and refactoring activities are heavily affected by multiplicity and estimation accuracy of thresholds that an antipattern requires. For this reason, we have experienced the influence of thresholds with respect to these two activities. In particular, we have conducted a sensitivity analysis by varying the numerical values of several thresholds of different performance antipatterns, on a case study. We have quantified threshold variations with the support of recall and precision metrics, and derived several useful findings for dealing with performance antipatterns on software models [31].

We summarize in the following the results that we have achieved so far:

(i) we have conducted a study aimed at highlighting the peculiar aspects of working on either the software side or the performance side [9];

(ii) we have preliminarily approached the problem of providing support to performance-based refactoring of software models, by introducing a first version of our antipattern-based approach [4]. In particular, we have defined a tailored RBML (i.e., a metamodel), with respect to a preliminary version of the PAML metamodel;

(iii) we have introduced further automation in our antipattern-based software refactoring process, by means of advanced MDE techniques (i.e., model differences, Higher-Order Transformations and Model-to-Model Transformations) [28];

(iv) we have defined a stable version of PAML, relying on READ to support the definition of model-based performance antipattern specifications and refactoring actions;

(v) we have experienced the influence of numerical thresholds on the capability and effectiveness of detecting and refactoring performance antipatterns, by analyzing the recall and precision of detection rules and refactoring actions respectively [31].

5 Expected contributions

The results obtained so far (as illustrated in Section 4) represent our contributions to the problem up to now. We are currently working on several directions, that shall outcome in the following expected contributions.

At the software side, the definition of mappings between PAML metamodel constructs and the ones of the considered software modeling languages [8,15,32]. Basing on this contribution, we will be able to provide automated MDE support to the specification, detection and refactoring of model-based performance antipatterns, independently from users modeling notations.

At the performance side, a method for the definition, the design and implementation of AQNs, and the analysis of the controlled system. The effects of this contribution will be twofold. It can be applied at design time to design controllers that will be implemented within the running system, and at runtime to induce in the running system the changes that are suggested from the controlled system.

6 Plan for evaluation and validation

In order to validate our work, we plan to provide running artifacts that support our framework at both the software and the performance side.

At the software side, we intend to develop an Eclipse plugin implementing a fully-automated sub-process for applying refactoring actions, based on model differencing, that will be evaluated by test users (e.g., students) by means of a large repository of refactoring actions, to demonstrate the expressivity of our approach and the significant support that we can provide in the model refactoring activity. Together with other tools by our research group (e.g., the tool for antipattern detection on software models), the evaluated plugin would provide support to the whole framework of Figure 1

At the performance side, we intend to realize a Modelica[1] library of components that define, through differential equations, adaptive QN elements with variable parameters, without requiring deep mathematical skills usually not mastered by software engineers. To validate the work at this side, we plan to use our library on several case studies, i.e., experimental sessions aimed at showing the behavior of a model and a controller under different scenarios, as described in Section 3.

7 Current status

To finalize our work, we identify the following tasks at the software and performance side, i.e., ST and PT respectively, that will lead to a Ph.D. dissertation (i.e., TD). Figure 2 shows a planned timeline for completion.

ST_1: To define mappings between PAML and the considered modeling languages, i.e., UML profiled with MARTE [8], PCM [15] and Æmilia [32].

ST_2: To define a graphical editor for creating Ecore-based PAML models in an easy way.

ST_3: To provide a large repository of refactoring actions in terms of (SRM,TRM) pairs that conform to PAML.

ST_4: To implement a fully-automated sub-process for applying refactoring actions (expressed in terms of models that represent differences between SRMs and corresponding TRMs), with the support of appropriate decision mechanisms [30].

PT_1: To realize a Modelica library of components that define (adaptive) QN elements with variable parameters, e.g., workload and operational profile.

PT_2: To study the synthesis of controllers for software and hardware adaptation that provide performance guarantees, and to extend our Modelica library to deal with specific "classes" of QNs.

[1] Modelica is an analysis and simulation environment to define dynamic models of engineered systems and to support the design and synthesis of suitable controllers [33].

PT_3: To experiment the use of our Modelica library on several case studies.
TD: Ph.D. thesis and dissertation.

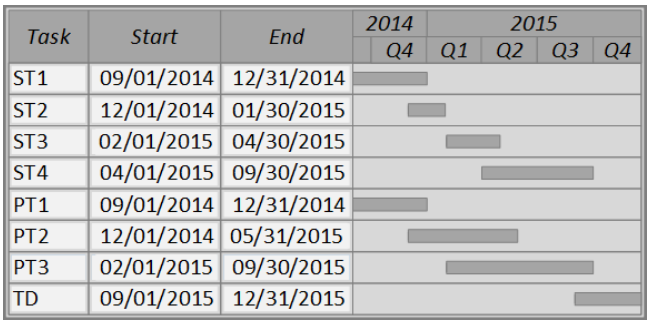

Fig. 2: Planned timeline w.r.t. the identified tasks.

Acknowledgements. This work has been supported by the European Office of Aerospace Research and Development (EOARD), Grant/Cooperative Agreement (Award no. FA8655-11-1-3055).

References

1. C. U. Smith, "Introduction to software performance engineering: Origins and outstanding problems," in *SFM*, vol. 4486 of *LNCSe*, pp. 395–428, Springer, 2007.
2. A. Martens, H. Koziolek, S. Becker, and R. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in *Proceedings of ICPE*, WOSP/SIPEW '10, (New York, NY, USA), pp. 105–116, ACM, 2010.
3. F. Ballesteros, F. Kon, M. Patio, R. Jimnez, S. Arvalo, and R. Campbell, "Batching: A design pattern for efficient and flexible client/server interaction," in *Transactions of PLP I* (J. Noble and R. Johnson, eds.), vol. 5770 of *LNCS*, pp. 48–66, Springer Berlin Heidelberg, 2009.
4. D. Arcelli, V. Cortellessa, and C. Trubiani, "Antipattern-based model refactoring for software performance improvement," in *ACM SIGSOFT QoSA*, pp. 33–42, 2012.
5. E. Lazowska, J. Kahorjan, G. S. Graham, and K. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.
6. C. M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (PUMA)," in *WOSP*, pp. 1–12, ACM, 2005.
7. G. Casale and G. Serazzi, "Quantitative system evaluation with java modeling tools," in *WOSP/SIPEW ICPE*, pp. 449–454, 2011.
8. Object Management Group (OMG), "UML Profile for MARTE," 2009. OMG Document formal/08-06-09.
9. D. Arcelli and V. Cortellessa, "Software model refactoring based on performance analysis: better working on software or performance side?," in *FESCA*, vol. 108 of *EPTCS*, pp. 33–47, 2013.
10. R. Eramo, V. Cortellessa, A. Pierantonio, and M. Tucci, "Performance-driven architectural refactoring through bidirectional model transformations," in *QoSA*, pp. 55–60, 2012.
11. T. Mens and G. Taentzer, "Model-driven software refactoring," in *WRT*, pp. 25–27, 2007.
12. H. Harreld, "NASA Delays Satellite Launch After Finding Bugs in Software Program," 1998.
13. V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Digging into UML models to remove performance antipatterns," in *ICSE Workshop Quovadis*, pp. 9–16, 2010.

14. C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *ICPE*, pp. 19–30, 2011.
15. S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.
16. V. Cortellessa, M. De Sanctis, A. Di Marco, and C. Trubiani, "Enabling performance antipatterns to arise from an adl-based software architecture," in *WICSA/ECSA*, pp. 310–314, 2012.
17. C. U. Smith and L. G. Williams, "More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot," in *ICMGC*, pp. 717–725, 2003.
18. 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.
19. T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," *Journal of Object Technology*, vol. 7, no. 3, pp. 55–90, 2008.
20. J. C. Doyle, B. A. Francis, and A. R. Tannenbaum, *Feedback Control Theory*. Macmillan, New York, 1992.
21. A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pp. 283–292, Nov 2011.
22. M. Maggio, H. Hoffmann, M. Santambrogio, A. Agarwal, and A. Leva, "Power optimization in embedded systems via feedback control of resource allocation," *Control Systems Technology, IEEE Transactions on*, vol. 21, pp. 239–246, Jan 2013.
23. A. Gambi, G. Toffetti, C. Pautasso, and M. Pezze, "Kriging controllers for cloud applications," *IEEE Internet Computing*, vol. 17, no. 4, pp. 40–47, 2013.
24. C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley, "Performance specifications and metrics for adaptive real-time systems," in *RTSS'10*, pp. 13–23, IEEE Computer Society, 2000.
25. G. Franks, D. C. Petriu, C. M. Woodside, J. Xu, and P. Tregunno, "Layered bottlenecks and their mitigation," in *QEST*, pp. 103–114, 2006.
26. J. Xu, "Rule-based automatic software performance diagnosis and improvement," *Performance Evaluation Journal*, vol. 67, no. 8, pp. 585–611, 2010.
27. V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," *SoSyM Journal*, 2012.
28. D. Arcelli, V. Cortellessa, and D. D. Ruscio, "Applying model differences to automate performance-driven refactoring of software models," in *EPEW*, vol. 8168 of *LNCS*, pp. 312–324, Springer, 2013.
29. R. B. France, S. Ghosh, E. Song, and D.-K. Kim, "A Metamodeling Approach to Pattern-Based Model Refactoring," *IEEE Software*, vol. 20, no. 5, pp. 52–58, 2003.
30. V. Cortellessa, A. Martens, R. Reussner, and C. Trubiani, "A process to effectively identify guilty performance antipatterns," in *FASE'10*, pp. 368–382, Springer-Verlag, 2010.
31. D. Arcelli, V. Cortellessa, and C. Trubiani, "Experimenting the influence of numerical thresholds on model-based detection and refactoring of performance antipatterns.," *ECEASST*, vol. 59, 2013.
32. A. Aldini, M. Bernardo, and F. Corradini, *A Process Algebraic Approach to Software Architecture Design*. Springer Publishing Company, Incorporated, 1st ed., 2009.
33. Modelica Association, "Modelica - A Unified Object-Oriented Language for Systems Modeling," 2012.