Approaching the model-driven generation of feedback to remove software performance flaws

Vittorio Cortellessa, Antinisca Di Marco, Romina Eramo, Alfonso Pierantonio, Catia Trubiani
Dipartimento di Informatica, University of L'Aquila
Via Vetoio,1 - 67010 Coppito (AQ) - Italy
{*vittorio.cortellessa,antinisca.dimarco,romina.eramo,alfonso.pierantonio,catia.trubiani*}*@univaq.it*

***Abstract*—The problem of interpreting results of performance analysis and providing feedback on software models to overcome performance flaws is probably the most critical open issue in the field of software performance engineering. Automation in this step would help to introduce performance validation as an integrated activity in the software lifecycle, without dramatically affecting the daily practices of software developers. In this paper we approach the problem with model-driven techniques, on which we build a general solution. Basing on the concept of performance antipatterns, that are bad practices in software modeling leading to performance flaws, we introduce metamodels and transformations that can support the whole process of flaw detection and solution. The approach that we propose is notation-independent and can be embedded in any (existing or future) concrete modeling notation by using weaving models and automatically generated model transformations. Finally, we discuss the issues opened from this work and the future achievements that are at the hand in this domain thanks to model-driven techniques.**

***Keywords*-Software Performance, Antipattern, Model-Driven Engineering, Model Transformation**

I. INTRODUCTION

Since more than a decade the problem of modeling and analyzing software performance from the beginning of the lifecycle has been tackled with a new type of approaches. The need of automation in the generation of performance models from software artifacts has gained a core role in the whole domain, because automation emerged as a key factor to overcome problems such as short time to market and specific skills required to build trustable models.

Numerous approaches have been introduced to automatically transform software artifacts (such as UML [2] models) into performance models (such as Petri Nets) [4]. Therefore the automated generation of performance models can be considered today as a quite mature discipline in the software performance domain [4], whereas other problems are taking the scene as key points for a complete automation of mechanisms in this domain.

At a certain point of the software lifecycle some typical steps (schematically represented in Figure 1) must be executed in order to run a complete performance modeling and analysis process. Rounded boxes in the figure represents operational steps whereas square boxes represent input/output data.

Arrows numbered from 1 through 4 represent the forward path from an (annotated) software model all the way through the production of performance indices of interest. While in this path quite well-founded approaches have been introduced for inducing automation in all steps [4], there is a clear lack of automation in the backward path that shall bring the analysis results back to the software model.

The main step of the backward path is the *result interpretation and feedback generation*. In Figure 1 all arrows labeled as 5 represent the possible inputs to the core step, that are: performance indices (label 5.a), and (annotated) architectural model (label 5.b). Problems in the architectural model are searched on the bases of these information. In this step the performance indices obtained from the model solution have to be interpreted in order to detect, if any, performance flaws. Once performance flaws have been detected (with a certain accuracy) somewhere in the model, solutions have to be applied to remove them. Typical solutions consist in design alternatives, namely feedback [25], that modify the original software model to achieve better performance. It is obvious that if all performance requirements are satisfied then the feedback simply suggests no changche e on the software model.

Strategies to drive the search of performance flaws and to generate feedback on a software model can take advantage from different techniques and can base on quite different elements that may depend on the adopted model notation, on the application domain, on environmental constraints, etc. Very promising instruments for flaws removal are *performance antipatterns* [23].

Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems [6]. They are known as antipatterns because their use (or misuse) may produce negative consequences. Antipatterns document common mistakes (i.e. the "bad practices") made during software development as well as their solutions: what to avoid and how to solve the problems. In particular, performance antipatterns are the ones that, if present in a model, may induce performance problems.

Performance problems are not necessarily originated by well-formalized antipatterns. In these cases an automated process is harder to achieve and they are typically handled by performance experts through the analysis of performance indices. However, performance antipatterns represent promising instruments that can sensibly improve the whole process of analysis and solution of performance issues in software systems.

In this paper we describe how to approach the interpretation of results and generation of feedback with model-

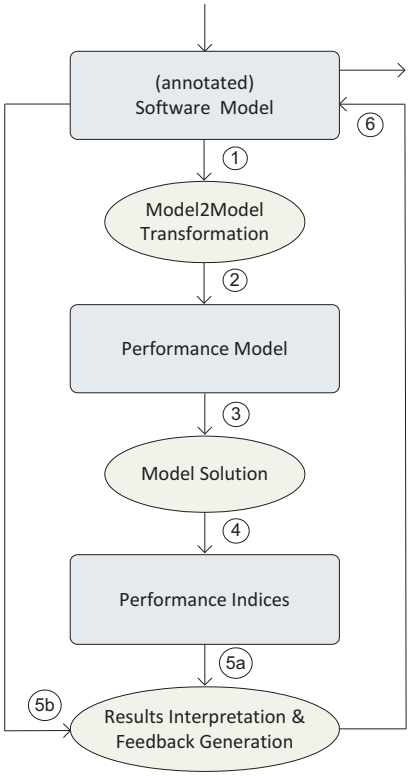

Figure 1. Automated software performance modeling and analysis.

driven techniques (i.e. the bottommost operational step in Figure 1). Performance antipatterns play a crucial role in this approach that is essentially made of three steps: antipattern specification, antipattern detection and antipattern solution. We show how model-driven techniques can be used to build a notation-independent approach that addresses all the above steps. We also show how this approach can be automatically embedded into any concrete modeling notation.

The paper is organized as follows: Section II introduces an informal definition of performance antipatterns, Section III describes the proposed model-driven framework, in Section IV some related work is presented, whereas in Section V we discuss relevant issues related to the proposed approach, finally Section VI concludes the paper.

II. Performance antipatterns and feedback

Results interpretation and feedback generation step is a critical activity of the software performance analysis process described in Figure 1. It embraces the localization of performance flaws in the software model starting from indices obtained by performance analysis, and their removal without violating specifications and constraints of the software design.

The search of performance flaws may be quite complex and needs to be smartly driven towards the problematic areas of the model. The complexity of this step stems from several factors: (i) performance indices are basically numbers associated to model entities and often they have to be jointly examined to let problems emerge; (ii) performance indices can be estimated at different levels of granularity and incomplete information often results from the model evaluation because it is unrealistic to keep under control all indices at all levels of abstraction; (iii) software models can be quite complex, they involve different characteristics of a software system (such as static structure, dynamic behavior, etc.), and performance problems sometimes appear only if those characteristics are cross-checked.

Search and removal of performance flaws can greatly benefit from a systematic automated management of performance antipatterns. The most interesting antipatterns are independent on the notations used to represent software and performance models. This antipattern characteristic allows to work only on those occurring in any model, because they are not related to any specific characteristic of modeling notations. For a similar reason performance antipatterns of our interest have not to be specific of any application domain.

An antipattern definition must include the problem (i.e. the model properties that characterize an antipattern) and the solution (i.e. the actions to be taken to remove the problem). Hence it may be beneficial to steer the searching process using performance problems, like antipatterns, that have well-known solutions to be adopted. However, up today performance antipatterns have been only informally defined, whereas a more solid formalism would be necessary to introduce automated techniques for their management.

The main source of performance antipatterns is the work done across years by Smith and Williams that have ultimately defined a number of 14 notation- and domain-independent antipatterns. Few other papers present additional performance antipatterns defined across different technologies, but they are not as general as the ones defined by Smith and Williams (see Section IV for more references).

In Table I some of the performance antipatterns defined are listed. For sake of space we do not explicitly comment each antipattern of Table I. Readers interested to more details can refer to [23]. Each row of Table I represents a specific antipattern that is characterized by three field (one per column) that are: antipattern name, problem description, solution description.

As shown in the leftmost part of Table I, we have grouped antipatterns in two different categories: one collects antipatterns that can be detected taking a snapshot of the software system, and they are referred in this paper as *Snapshot-based* Performance Antipatterns; the other one collects those for which a single snapshot is not enough to capture the performance problems induced in the software system, and they are referred as *Monitoring-based* Performance Antipatterns because their detection must lay on system data collected along a certain interval of time. The only difference is that for the Snapshot-based ones every element identifier does not need to bring more than one value (i.e. the observed value of the corresponding element at a fixed instant of time), whereas

	Antipattern		Problem	Solution
Snapshot-based	Blob (or god class/component)		Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the applications data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the applications top-level classes, and to keep related data and behavior together.
	Unbalanced Processing	Concurrent Processing Systems	Occurs when processing cannot make use of available processors.	Restructure software or change scheduling algorithms to enable concurrent execution.
		"Pipe and Filter" Architectures	Occurs when the slowest filter in a pipe and filter architecture causes the system to have unacceptable throughput.	Break large filters into more stages and combine very small ones to reduce overhead.
		Extensive Processing	Occurs when extensive processing in general impedes overall response time.	Move extensive processing so that it doesnt impede high traffic or more important work.
	[. . .]		[. . .]	[. . .]
Monitoring-based	Traffic Jam		Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.	Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.
	The Ramp		Occurs when processing time increases as the system is used.	Select algorithms or data structures based on maximum size or use algorithms that adapt to the size.
	[. . .]		[. . .]	[. . .]

Table I
PERFORMANCE ANTIPATTERNS: PROBLEM AND SOLUTION [23].

for the Monitoring-based ones some identifiers may need to assume multiple value (i.e. the observed values of the corresponding element along a certain interval of time).

Such classification can be extended to include technology-dependent performance antipatterns on the basis of a specific principle they reveal. For example the Bloated Session Antipattern[10] describes a situation in EJB systems where a session bean has become too bulky: such session bean generally implements methods that operate against a large number of abstractions.

Basing on a more formal specification of performance antipatterns, the *results interpretation and feedback generation* step of the process in Figure 1 can be split in two sub-steps: antipattern detection and antipattern solution, where the latter may produce several design alternatives the developer must analyze to adopt the most appropriate one(s).

It is worth to note that we are acting in a non-functional domain, therefore the process is unavoidably based on heuristic evaluations and decisions, as there is no guarantee that the feedback provided deterministically solves the problems. Hence the forward path has to be taken again, starting from the updated software model and ending up with new performance indices that shall confirm the performance flaws have been actually removed. Only after this validation the software lifecycle can proceed. If the validation is unsuccessful the backward path has to be run again and the whole process restarts.

In the following section, we discuss how we intend to implement the antipattern specification, detection and solution via model-driven techniques that are scalable and customizable to specific software modeling notations. Indeed, we introduce a general framework that grounds on a system modeling language independent from concrete notations adopted for software modeling and inspired to languages generally used to model a software system, enriched with terms related to performance aspects.

III. MODEL-DRIVEN APPROACH FOR ANTIPATTERN MANAGEMENT

Model Driven Engineering [22] (MDE) leverages intellectual property and business logic from source code into high-level specifications enabling more accurate analyses. In general, an application domain is consistently analyzed and engineered by means of a *metamodel*, i.e., a coherent set of interrelated concepts. A model is said to *conform* to a metamodel, or in other words it is expressed by the concepts encoded in the metamodel, constraints are expressed at the meta-level, and model transformations are based on source to target metamodels.

As any other software artifacts, models are subject to the evolutionary pressure due to the necessity of accommodating extensions and unforeseen requirements [13], [16]. In the performance software analysis, evolution takes place through changes made in models in order to achieve better performance. By generating performance models and related indices, problems can be better understood and feedbacks can drive the refactoring of the original

software model in order to improve its performance. As aforementioned antipatterns are of crucial relevance as they include the definition of the problem and its solution.

This work proposes a framework for modeling performance antipatterns. The approach is general and makes use of parametric notations that can be instantiated with any modeling language. In practice, there are no constraints for the concrete modeling language to use with our approach (e.g., it can be a stochastic notation such as Petri Nets). Every antipattern is formalized by assuming the definition of certain system properties that can be deterministic or probabilistic, and the detection of the occurrence of an antipattern is based on the satisfaction of such properties.

A. Specifying antipatterns

This section proposes a modeling language able to specifing antipattern. The AntiPattern Modeling Language (APML) is a metamodel specifically tailored to describe performance problems and their solutions. It consists of a number of constituent languages as illustrated in Figure 2: *a)* the Software Modeling Language (SML), a general-purpose language which encompasses the minimal amount of modeling elements to describe a software system; *b)* an enrichment of SML (SML+) including the performance parameters; and finally *c)* a Refactoring Modeling Language (RML) for formalizing the solutions in terms of refactorings, i.e., modifications the original model must undergo to overcome the performance problems.

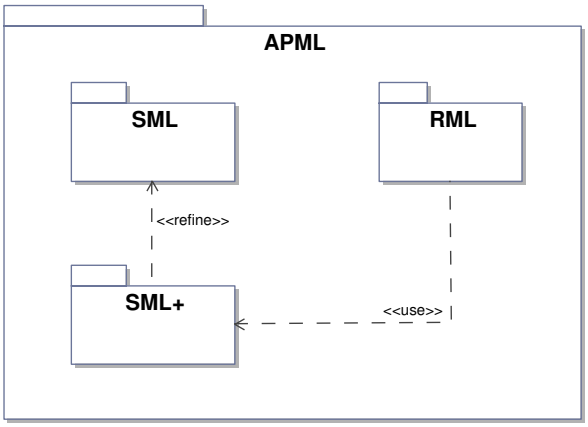

Figure 2. AntiPattern Modeling Language (APML) Structure.

An antipattern is a model conforming to APML and specifies a potential problem and its solution: the former consists of a *static*, *dynamic* and *deployment* view, each of them conforming to the SML+ metamodel; whereas the latter is a *refactoring* conforming to RML. The (multiple) occurrence of views in a software model may represent a problem and requires the corresponding refactoring to be applied. Each antipattern model is intended to formalize an entry in Table I.

B. Embedding the approach in concrete modeling notations

The antipattern metamodel is defined on the basis of the SML+ language, which is the parametric part of APML and contains only the minimal amount of concepts that are essentials for specifying antipatterns. A parameter passing mechanism based on weaving models [5] is realized by mapping the concepts in SML+ into the corresponding concepts in the actual modeling language (as done in [15] for different purposes, though). Weaving models represent a useful technique in software modeling, as they can be used for setting fine-grained relationships between models or metamodels and executing operations on them based on the link semantics. This is important to make the approach agnostic of the specific modeling notation which is then used to specify annotated system models (e.g., UML + MARTE [3], Stochastic Process Algebras, Æmilia ADL, etc.).

For instance, if the modeling language given by UML and the MARTE profile has been chosen to model the software system, then it can be used in place of SML+ by giving a weaving model as in Figure 3, which explicitly defines the correspondences among the two metamodels. Like this, the concepts in SML+ will be substituted by those in UML+MARTE and the antipatterns given in APML can be automatically translated into notations which refer to the specific model elements which are proper of UML+MARTE.

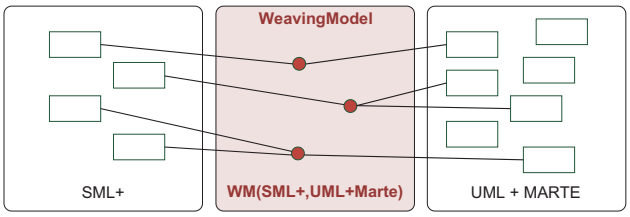

Figure 3. Metamodel instantiation via weaving models

As said, antipatterns do not refer to any specific notation and it is important to specify them according to a neutral notation which is agnostic of the concrete modeling languages. Once designers understood and defined the explicit correspondences between the parts, the weaving model does not interfere with the definition of the models on either side, achieving a clear separation of them and their connections. The task of the design is to understand and define the explicit links between the parts.

As any other model, weaving models can be used in automated transformations to generate other artifacts. In fact, it is possible to have higher-order transformations (HOT) which starting from the weaving models are able to generate other metamodel-specific transformations which, in turn, *embeds* the antipattern in the actual modeling language, as depicted in Figure 4. Additional HOTs can be given to perform other transformations as described in the next section.

C. Detecting antipatterns

As previously discussed, providing the specification of a problem based on the modeling language is of important relevance to locate antipatterns in software models. In general, model element patterns can be seen as model

queries, i.e., diagrammatic notations which corresponds to first-order predicates, which can be naturally expressed by means of OCL [18] expressions. Therefore, each model conforming to APML can be given an OCL-based semantics in a similar way to the approach in [24], which interrogates the elements of a system model to detect antipatterns and eventually to give them a solution. Especially, having an antipattern expressed in APML is convenient as long as we are able to translate it into a concrete modeling language, e.g., UML+MARTE, as describe above by means of weaving models.

The overall approach is depicted in Figure 5. An antipattern model AP once translated into its concrete modeling language (e.g., UML+MARTE) counterpart AP′ can, in turn, be transformed in an OCL expression ⟦AP′⟧ by means of another HOT. Additionally, a collection of auxiliary function AF can also be generated always accordingly to the concrete metamodel to evaluate specific information on the models useful during the detection. A generated OCL predicate ⟦AP′⟧ is able to filter a software model M and detect eventual occurrences of the antipattern AP it originated from. Once a given antipattern AP is detected then the problem it underlies must be solved by applying the associated refactoring R_{AP} as described in the next section.

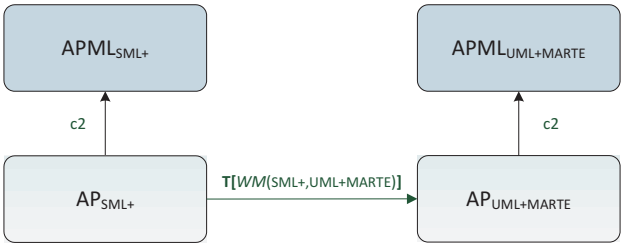

Figure 4. Weaving model and translation of the approach over other Software modeling language.

D. Resolving antipatterns

The specification of the resolution can be defined as a set of modification actions that can be applied on the system model in order to solve the detected performance problems. To this end, *difference models* can be used to represent modifications between subsequent versions of the model and, thus, refactorings.

The problem of model differences is intrinsically complex and requires specialized algorithms and notations to match the abstraction level of models [14]. Recently, in [7], [21] two similar techniques have been introduced to represent differences as models; interestingly these proposals combine the advantages of declarative difference representations and enable the reconstruction of the final model by means of automated transformations which are inherently defined in the approaches. In this respect, the proposals are adequate to support the refactoring representation as the collection of modifications between two subsequent versions of a model. In the proposed approach, the difference representation approach defined in [7] is used.

As already said, an antipattern consists of a pattern (whose occurrence in a software model is considered a bad practice) and a refactoring which possibly solve the performance problem. Therefore, if an antipattern AP is detected in a model M, then a the refactoring R_{AP} has to be applied. Similarly to the antipattern model, a HOT is needed to translate the refactoring in the concrete notation, e.g., UML+MARTE, to obtain, say $R_{AP'}$. The application of the refactoring to M produces the feedback M′ where the problems has been removed or neutralized. The application of the refactoring is based on model-driven techniques described in [7] and resembles a model patch functionality.

When several antipatterns are detected, the corresponding resolutions must be applied giving eventually place to occurrences of further antipatterns which have, in turn, to be resolved. The situation give place to a satisfiability problem, whose complexity is at least NP-complete and whose solution requires complex techniques. An interesting possibility may be represented by using the *critical pairs analysis* [17], which provides a mean to avoid conflicting and divergent *refactorings*.

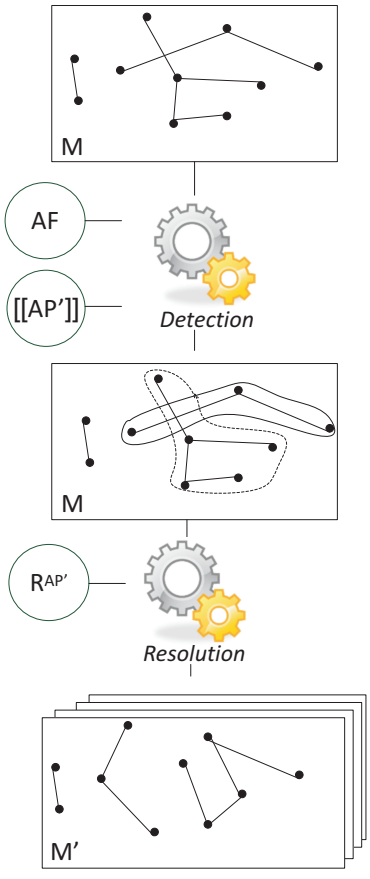

Figure 5. Feedback Generation Approach.

IV. Related work

A first proposal of automated generation of feedback due to the software performance analysis can be found in [9], where the detection of performance flaws is yet demanded to the analysis of Layered Queued Network

models and uses informal interpretation matrices as support.

The issue of detecting performance antipatterns has been addressed in [19], where a rule-based performance diagnosis tool, named Performance Antipattern Detection (PAD), is presented. However PAD only deals with Component-Based Enterprise Systems, targeting Enterprise Java Bean (EJB) applications. It is based on monitoring data from running systems, and it extracts the run-time system design and detects EJB antipatterns by applying rules to it. Therefore its scope is restricted to such domain, whereas in our approach the starting point is a notation-independent model of the software system.

Another interesting work on the software performance diagnosis and improvements has been recently proposed in [26]: rules to identify patterns of interaction between resources are defined and specified as Jess rules [1]. Performance flaws are identified before the implementation of the software system, even if they are related only on bottlenecks (e.g. the "One-Lane Bridge" antipattern in Smith's classification) and long paths. However, performance issues are identified at the level of the LQN performance model, and the translation of these model properties into design changes could hide some possible refactoring solutions. In this paper we refer both to performance and design features of the software system in the feedback generation process in order to maintain the whole information we need to choose the best design alternatives.

By taking a wider look out of the performance domain, the detection of *antipatterns* is a quite recent research topic, whereas there has already been a significant effort in the area of detecting software design *patterns*. We do not intend to address such wide area, but it is worth to mention a recent work that uses model-driven instruments to deal with patterns. In [11] a metamodeling approach to pattern specification and detection has been introduced. In the context of the OMG's 4-layer metamodeling architecture, the authors propose a pattern specification language (i.e. Epattern, at the M3 level) used to specify patterns (at the M2 level). The detection of design patterns is demanded to the generation of algorithms from Epattern.

Finally, we highlight that our model-driven approach is inspired by the mechanisms in [24] and [8]. [24] presents a graphical notation to specify selection queries on models based on UML by means of "Join Point Designation Diagrams" (JPDD). They aim to facilitate comprehension of query statements and estimation of the selected model elements. Inspired by this work, [8] proposes a domain-specific language able to define and manage conflicts caused by cooperative updates over the same model elements.

V. Discussion

As illustrated in this paper, model-driven techniques represent very promising instruments to tackle the problem of automating performance result interpretation and feedback generation. The scope of this work has been to provide a framework where single activities (e.g. detection and solution) interact each other through well-defined models (e.g. instances of antipattern metamodel) to achieve such goal.

However, the work presented here highlights critical pending issues (in addition to the ones discussed above in the paper) that have to be faced in order to automate the whole process. Most of these issues relate to the antipattern resolution activity, and we summarize them in this section.

Let us assume that the detection activity is based on brute force and barely produces a list of all antipatterns that have been detected in a certain software model. Now, the main problem at solution phase is to decide how many antipatterns to solve, which ones and in what order. To this goal, we are working to introduce a scoring process able to identify the most promisingly "guilty" performance antipatterns between the detected ones.

In fact, in order to build each design alternative a certain number of antipatterns has to be solved. Two main categories of problems can be defined in this phase: requirement problems and coherency problems.

Requirement problems raise when one or more antipatterns cannot be solved due to pre-existing (functional or non-functional) requirements. Example of functional requirements may be legacy components that cannot be split and re-deployed whereas the antipattern solution consists of these actions. Example of non-functional requirements can be budget limitations that do not allow to adopt an antipattern solution due to its extremely high cost. Many other examples can be provided of requirements that (implicitly or explicitly) may affect the antipattern resolution activity. For sake of automation such requirements should be pre-defined so that the whole process can take into account them and preventively excluding infeasible solutions.

Coherency problems raise when (although feasible) feedback, represented as the solution of a certain number of antipatterns, cannot be unambiguously applied due to incoherencies among antipattern solutions. For example, the solution of one antipattern may suggest to split a component into three finer grain components, while another antipattern in the same proposed feedback suggests to merge the original component with another one. These two actions obviously contradict each other, although no pre-existing requirement limits their application. Even in cases of no explicit conflict between antipattern solutions, coherency problems can be raised from the order of application of solutions. In fact the result of the sequential application of two (or more) antipattern solutions is not guaranteed to be invariant with respect to the application order. Criteria have to be introduced to drive the application order of solutions in these cases. Similarly to requirement problems, the best scenario for sake of automation shall be to devise mechanisms acting during the whole process and aimed at preventively excluding incoherent solutions.

However, more lightweight approaches can be adopted for both categories of problems. They basically con-

sist in generating all possible alternatives coming from antipattern solutions, and thereafter proposing them to stakeholders for manual selection of different options.

VI. Conclusion

This work provides a structured approach, based on model-driven techniques, to the problem of interpreting results of performance analysis and generating feedback for software models.

In order to provide a general solution to this problem, our approach has been conceived as notation-independent, in that it does not lay on any characteristic of concrete modeling notations. To be applied in practice we have devised a mechanism based on weaving models and model transformations that allows to embed detection and solution techniques in any specific notation.

This is a first work in the direction of providing automation in the backward path from performance analysis to software modeling. Although much work must be done to validate the applicability of the whole approach in practice, on the basis of this design experience we retain that model-driven techniques represent a very promising ground on which sharp solutions to the problem can be built.

Multiple interesting research directions can be originated from this work, both in the fields of performance analysis and model-driven engineering, and we shortly mention the ones that we intend to pursue in a short-term view.

First of all, the metamodels introduced here have to be appropriately built, following the notation-independency criterion that represents an added value of the whole approach. Pivot modeling languages can greatly help in our process of construction of a language-independent metamodel. In fact, we intend to inherit from languages (such as KLAPER [12] and CSM [20]) the definitions that are useful to our metamodel. Once defined such metamodels, weaving models shall be introduced to illustrate the possibility of automatically embedding these general techniques into concrete modeling notations.

For what concerns performance, criteria to apply detection and solution of antipatterns have to be defined (as illustrated in Section V) while taking into account pre-existing requirements. On the model-driven engineering side, at the same time, techniques to solve conflicts in model transformations, or to decide the order of application of a sequence of transformations, have to be applied to this domain.

Acknowledgment

This work has been partly supported by the national Projects D-ASAP (Architetture Software Adattabili e Affidabili per Sistemi Pervasivi), and PACO (Performability-Aware Computing: Logics, Models, and Languages).

References

[1] JESS, the Rule Engine for the Java Platform, *http://www.jessrules.com/jess/index.shtml*.

[2] UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group, Inc. (2005), *http://www.omg.org/cgi-bin/doc?formal/05-07-04*.

[3] UML Profile for MARTE beta 2, OMG document ptc/08-06-09, Object Management Group, Inc. (2008), *http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf*.

[4] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[5] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[6] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1998.

[7] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007.

[8] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 311–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] V. Cortellessa and L. Frittella. A framework for automated generation of architectural feedback from software performance analysis. In *Proc. Formal Methods and Stochastic Models for Performance Evaluation*, 2007.

[10] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf. *J2EE Antipatterns*. 2003.

[11] M. Elaasar, L. C. Briand, and Y. Labiche. A metamodeling approach to pattern specification. In *Lecture Notes in Computer Science: Model Driven Engineering Languages and Systems*, volume 4199/2006, pages 484–498. Springer Berlin / Heidelberg, 2006.

[12] V. Grassi, R. Mirandola, E. Randazzo, and A. Sabetta. Klaper: an intermediate language for model-driven predictive analysis of performance and reliability. In *Common Component Modeling Example*, volume 5153 of *LNCS*, pages 327–356, 2008.

[13] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[14] Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA Work. MDSD*, 2004.

[15] I. Malavolta, H. Muccini, P. Pelliccione, and D. Tamburri. Providing architectural languages and tools interoperability through model transformation technologies. *IEEE Transactions on Software Engineering (TSE). IEEE computer Society*, 2009. to appear.

[16] T. Mens, J. Buckley, A. Rashid, and M. Zenger. Towards a taxonomy of software evolution. Technical Report vub-prog-tr-02-05, Vrije Universiteit Brussel, 2003.

[17] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electr. Notes Theor. Comput. Sci*, 127(3):113–128, 2005.

[18] Object Management Group (OMG). OCL 2.0 Specification, 2006. OMG Document formal/2006-05-01.

[19] T. Parsons and J. Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–90, 2008.

[20] D. Petriu and M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from uml designs. *Software and Systems Modeling (SoSyM)*, 6:163–184(22), June 2007.

[21] J. Rivera and A. Vallecillo. Representing and Operating with Model Differences. In *Procs of the 46th Int. Conf. TOOLS EUROPE 2008*, 2008.

[22] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[23] C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, 2003.

[24] D. Stein, S. Hanenberg, and R. Unland. A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In *MDAFA'03 and MDAFA'04*, volume 3599 of *LNCS*, pages 77–92, 2005.

[25] C. M. Woodside, M. Franks, and D. C. Petriu. The future of software performance engineering. In *FOSE'07*, pages 171–187, 2007.

[26] J. Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP'08, Workshop on Software Performance*, pages 1–12, 2008.