APPROACHES FOR SOFTWARE PERFORMANCE FEEDBACK

[1]Ms.Warsha M.Choudhari, [2]Mrs Rinku S. Ashtankar, [3]Ms.Shalini Kharkate
[1]Asst.Professor, [2] Asst. Professors, [3] Asst. Professors,
[1]Information Technology, [2]Computer Science & Engineering, [3]Computer Engineering
[1]Datta Meghe Institute of Engineering, Technology & Research, Wardha, India, [2]ITM College of Engineering, Nagpur, India, [3]Govt. Polytechnic, Gadchiroli, India

Abstract — Over the last decade, research has highlighted the importance of integrating the performance analysis in the software development process. Software Performance Engineering (SPE) has been recognized as the discipline that represents the entire collection of engineering activities, used throughout the software development cycle, and directed to meet performance requirements. Performance is in fact an essential quality attribute of every software system; it is a complex and a pervasive property difficult to understand. If performance targets are not met, a variety of negative consequences (such as damaged customer relations, business failures, lost income, etc.) can impact on a significant fraction of projects. Performance problems cause delays, failures on deployment, redesigns, even a new implementation of the system or abandonment of projects, which lead to significant costs. All these factors motivate the activities of modeling and analyzing the performance of software systems at the earlier phases of the lifecycle by reasoning on predictive quantitative. To provide an automated feedback to make the performance analysis results usable at the software architectural level. Results in order to avoid an expensive rework, possibly involving the overall software system.

***IndexTerms*: Software Architecture, Performance Evaluation, SPE.**

I. INTRODUCTION

In the software development process it is fundamental to understand if performance requirements are fulfilled, since they represent what end users expect from the software system, and their unfulfillment might produce critical consequences. The early development phases may heavily affect the quality of the final software product, and wrong decisions at early phases may imply an expensive rework, possibly involving the overall software system. Therefore, performance issues must be discovered early in the software development process, thus to avoid the failure of entire projects.

The model-based approach, pioneered under the name of Software Performance Engineering (SPE) creates performance models early in the development cycle and uses quantitative results from these models to adjust the architecture and design with the purpose of meeting performance requirements. Software architectures have emerged as a foundational concept for the successful development of large, complex systems, since they support five aspects of the software development: understanding, reuse, evolution, analysis and management, A software architectural model Complementary types of model provide different system information. Such different models present the system from different perspectives, such as external perspective showing the system's context or environment, behavioral perspective showing the behavior of the system, etc. We refer to (annotated) models, since annotations are meant to add information that led to execute performance analysis such as the incoming workload to the system, service demands, hardware characteristics, etc. There exist many notations to describe all these aspects of a software system (e.g. automata, process algebras, and petrinets and process algebras).

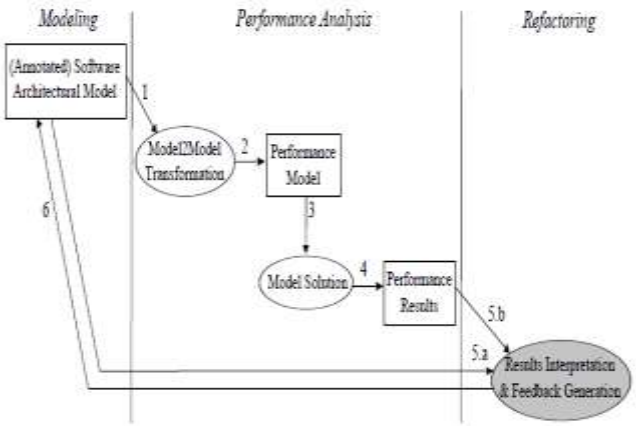

Fig. Automated software performance process

II. ANTIPATTERN-BASED APPROACHES

The term Antipatterns appeared for the first time in contrast to the trend of focus on positive and constructive solutions. Differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences. Antipatterns have been applied in different domains. For example, in data-flow antipatterns help to discover errors in workflows and are formalized through the CTL* temporal logic. Performance Antipatterns, as the name suggests, deal with performance issues of the software systems. They introduced the PASA (Performance Assessment of Software Architectures) approach. It aims at achieving good performance results through a deep understanding of the architectural features. This is the approach that firstly introduces the concept of antipatterns as support to the identification of performance problems in software architectural models as well as in the formulation of architectural alternatives. However, this approach is based on the interactions between software architects and performance experts; therefore its level of automation is still low.

III. RULE-BASED APPROACHES

Barber et al. in [2] introduced heuristic algorithms that in presence of detected system bottlenecks provide alternative solutions to remove them. The heuristics are based on architectural metrics that help to compare different solutions. In a Domain Reference Architecture (DRA) the modification of functions and data allocation can affect non-functional properties (for example, performance-related properties such as component utilization).

The tool RARE guides the derivation process by suggesting allocations based on heuristics driven by static architectural properties. The tool ARCADE extends the RARE scope by providing dynamic property measures. ARCADE evaluation results subsequently fed back to RARE can guide additional heuristics that further refine the architecture. However, it basically identifies and solve only software bottlenecks, more complex problems are not recognized.

Dobrzanski et al. in [7] tackled the problem of refactoring UML models. In particular, bad smells are defined as structures that suggest possible problems in the system in terms of functional and non-functional aspects. Refactoring operations are suggested in the presence of bad smells. Rules for refactoring are formally defined, and they take into account the following features:

(i) Cross integration of structure and behavior;
(ii) Support for component-based development via composite structures; and
(iii) Integration of action semantics with behavioral constructs.

However, no specific performance issue is analyzed, and refactoring is not driven by unfulfilled requirements.

IV. SEARCH-BASED APPROACHES

A wide range of different optimization and search techniques have been introduced in the field of Search-Based Software Engineering (SBSE) [3, 4], i.e. a software engineering discipline in which search-based optimization algorithms are used to address problems where a suitable balance between competing and potentially conflicting goals has to be found.

Two key ingredients are required: (i) the representation of the problem;
(ii) the definition of a fitness function.

In fact, SBSE usually applies to problems in which there are numerous candidate solutions and where there is a fitness function that can guide the search process to locate reasonably good solutions. A suitable representation of the problem allows to automatically exploring the search space for the solutions that best fit the fitness function that drives towards the sequence of the refactoring steps to apply to this system (i.e. altering its architectural structure without altering its semantics).

In the software performance domain both the suitable representation of the problem and the formulation of the fitness function are not trivial tasks, since the performance analysis results are derived from many uncertainties like the workload, the operational profile, etc. that might completely modify the perception of considering candidate solutions as good ones. Some assumptions can be introduced to simplify the problem and some design options can be explicitly defined in advance to constitute the population [6] on which search based optimization algorithms apply. However, we believe that in the performance domain it is of crucial relevance to find a synergy between the search techniques that involve the definition of a fitness function to automatically capture what is required from the system, and the antipatterns that might support such function with the knowledge of bad practices and suggest common solutions, in order to quickly converge towards performance improvements.

In fact, as recently outlined in [5], there is a mutually beneficial relationship between SBSE and predictive models. In particular eleven broad areas of open problems (e.g. balancing functional, nonfunctional properties of predictive models) in SBSE for predictive modeling.

4.1 DESIGN SPACE EXPLORATION APPROACHES

Zheng et al. in [8] described an approach to find optimal deployment and scheduling priorities for tasks in a class of distributed real-time systems. In particular, it is intended to evaluate the deployment of such tasks by applying a heuristic search strategy to LQN models. However, its scope is restricted to adjust the priorities of tasks competing for a processor, and the only refactoring action is to change the allocation of tasks to processors. Bondarev et al. in [12] proposed a design space exploration methodology, i.e. DeSiX (DEsign, SImulate, eXplore), for software component-based systems. It adopts multidimensional quality attribute analysis and it is based on:

(i) various types of models for software components, processing nodes, memories and bus links,
(ii) scenarios of system critical execution, allowing the designer to focus only on relevant static and dynamic system configurations,
(iii) simulation of tasks automatically reconstructed for each scenario, and
(iv) Pareto curves [13] for identification of optimal architecture alternatives.

4.2 METAHEURISTIC APPROACHES

Canfora et al. in [11] used genetic algorithms for Quality of Service (QoS)-aware service composition, i.e. to determine a set of concrete services to be bound to the abstract ones in the workflow of a composite service. However, each basic service is considered as a black-box element, where performance metrics are fixed to a certain unit (e.g. cost=5, resp. time=10), and the genetic algorithms search the best solutions by evaluating the composition options. Hence, no real feedback (in terms of refactoring actions in the software architectural model such as split a component) is given to the designer, with the exception of pre-defined basic services. Aleti et al. in [10] presented a framework for the optimization of embedded system architectures. In particular, it uses the AADL (Architecture Analysis and Description
Language) [9] as the underlying architecture description language and provides plug-in mechanisms to replace the optimization engine, the quality evaluation algorithms and the constraints checking. Architectural models are optimized with evolutionary algorithms considering multiple arbitrary quality criteria. However, the only refactoring action the framework currently allows is the component re-deployment.

V. CONCLUSION:

The performance knowledge that we have organized for reasoning on performance analysis results can be considered as an application of data mining to the software performance domain. It has been grouped around design choices and performance model analysis results concepts, thus to act as a data repository available to reason on the performance of a software system. Performance antipatterns have been of crucial relevance in this context since they represent the source of the concepts to identify performance flaws as well as to provide refactoring in terms of architectural alternatives

REFERENCES

[1] Olabiyisi S.O, Omidiora E.O, Uzoka F.M.E, Victor Mbarika and Akinnuwesi B.A, "A Survey of Performance Evaluation Models for Distributed Software System Architecture" Proceedings of the World Congress on Engineering and Computer Science 2010 Vol I WCECS 2010, October 20-22, 2010, San Francisco, USA

[2] Barber, K. S., Graser, T. J., And Holt, J. "Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation", In ASE (2002), pp. 172–182.

[3] Harman, M. "The Current State and Future of Search Based Software Engineering", In Fose (2007), pp. 342–357.

[4] Harman, M., Mansouri, S. A., And Zhang, Y. "Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications", Tech. Rep. TR-09-03, 2009.

[5] Harman, M. "The relationship between search based software engineering and predictive modeling. In Proceedings of the 6th International Conference on Predictive Models in Software Engineering", (New York, NY, USA, 2010), ACM, pp. 1:1– 1:13.

[6] Harman, M. "Why the Virtual Nature of SoftwareMakes It Ideal for Search Based Optimization", In Rosenblum and Taentzer [114], pp. 1–12.

[7] Dobrzanski, L., And Kuzniarz, L., "An approach to refactoring of executable UML models", In ACM Symposium on Applied Computing (SAC) (2006), pp. 1273– 1279.

[8] Zheng, T., And Woodside, C. M. "Heuristic optimization of scheduling and allocation for distributed systems with soft deadlines", In Computer Performance Evaluation / TOOLS (2003), P. Kemper and W. H. Sanders, Eds., vol. 2794 of Lecture Notes in Computer Science, Springer, pp. 169–181.

[9] Ehrgott, M. "Multicriteria Optimization", 2005.

[9] Feiler, P. H., Gluch, D. P., And Hudak, J. J. "The Architecture Analysis and Design Language (AADL): An Introduction. Tech. Rep.", CMU/SEI-2006-TN-001, Software Engineering Institute, Carnegie Mellon University, 2006.

[10] Aleti, A., Bj¨Ornander, S., Grunske, L., And Meedeniya, I. ArcheOpterix: "An extendable tool for architecture optimization of AADL models", In MOMPES (2009), pp. 61–71.

[11] Omitaomu A. Oluwafem i and Adedeji Badiru, 2007. "Fuzzy Present Value Analysis Model for Evaluating Information System Projects", Published in the Engineering Economist, Vol. 52, Issue 2, pp 157 – 178.

[12] Bailey H. David and Snav ely Allan, 2005. "Performance Modeling: Understanding the Present and Predicting the Future", Proceedings of Euro-Par, Lisbon, Portugal.

[13] Dwyer B. Mathew, Hatcliff John, Pasareanu S. Corina and Visser Willen, 2007. Formal Software Analysis: Emerging Trends in Software Model Checking. Future of Software Engineering (FOSE'07). Copyright IEEE.