# Automated generation of structured feedback from software performance analysis

[1]Kapil Kumar Verma, [2]Anil Kumar Solanki

[1] Research Scholar, [2]Professor

[1]Department of CSE, Bhagwant University, Ajmer, INDIA,

[2]CSE Department, BIET, Jhansi, INDIA.

***Abstract :*** A rather complex task in the performance analysis of software architectures has always been the interpretation of the analysis results and the generation of feedback that may help developers to improve their architecture with alternative "better performing" solutions. This is due, on one side, to the fact that performance analysis results may be rather complex to interpret (e.g., they are often collections of different indices) and, on the other side, to the problem of coupling the "right" architectural alternatives to results, that are the alternatives that allow to improve the performance by resolving critical issues in the architecture. In this paper we propose a framework to interpret the performance analysis results and to propose alternatives to developers to improve their architectural designs. The interpretation of results is based on the ability to automatically recognize performance anti-patterns in the software architecture**.**

**Keywords:** Software Performance, Layered Queuing Networks, Architectural feedback, Performance indices.

## I. INTRODUCTION

The validation of software performance often finds obstacles to be accepted as a daily practice in the software development processes for many reasons. One of the major drawbacks is the lack of automated support. The performance validation activity can be summarized in four main steps: generation of a performance model from a software model, performance model analysis, interpretation of analysis results, generation of feedback on the software model. Among the above steps, the analysis of a performance model (e.g. a Petri Net) is the one that has been studied since more time and for which well assessed techniques exist [5]. In the last few years many efforts have been devoted to introduce automation in the first step that is the performance model generation. Several methodologies and tools have been introduced to transform a software model (e.g., a set of UML diagrams) into a performance model (e.g. a Queuing Network) [1].

However, in order to close the 4-steps loop described above, automation shall be introduced in the last few steps that represent the reverse path from the performance model to the software model. What obviously software developers expect from performance analysis is not a repository of values and curves that represent different indices (such as throughput, utilization, etc.) at different level of granularity, and that are very hard to decipher even by performance experts. They would expect to receive an interpretation of these results in terms of directives, suggestions, architectural alternatives that can drive their development process towards a software product able to meet the performance requirements.

With the support of automated tool their decision about the software architecture (and later decisions) could be driven even by performance issues that, instead, are often discovered at the end of the process when changes are much more expensive to be made.

Goal of this paper is to introduce a process that can drive the performance result interpretation and the generation of structured feedback. The rationale of our process founds on three main considerations: (i) performance analysis is a hierarchical task that, in order to produce feedback, often must investigate tiny details of the system architecture; for this reason, each iteration of our process lays on a zooming approach that, from system-level performance indices, drives down to resource/component-level indices; (ii) only a structured and integrated knowledge may lead to produce significant feedback; for this reason, the core data used in our process have been organized in matrices that are shared by the interpretation and the generation phases; (iii) for a hierarchical investigation, it plays a crucial role the capability to recognize architectural patterns that may adversely affect the system performance; for this reason, we have classified and solved a set of patterns that can be recognized with simple pattern matching techniques.

## II. LITERATURE REVIEW

In [12] the PASA (Performance Analysis of Software Architecture) approach has been introduced that aims at achieving good performance results through a deep understanding of the architectural features. This is the approach that better define the concept of antipattern that will be widely used in our approach. However, this approach is based on the interactions between software architects and performance experts; therefore its level of automation is quite poor.

A simulation based approach has been introduced in [8], where the model simulation produces data on the system states that, once processed, can offer useful suggestions about the maximum performance achievable with the current system configuration.

A quite interesting work has been introduced in [3], where "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 presence of "bad smells". Rules for refactoring are formally defined.

## III. Automated generation of feedback:

In this section we illustrate our approach for the interpretation of performance results and the automated generation of architectural alternatives. The approach goes through two fundamental phases:

An identification phase (or interpretation phase), where the analysis of the performance results brings to identify particular scenarios that affect performance;

A construction phase (or generation phase), where several architectural alternatives are constructed, basing on the information collected in the previous phase.

Even though these two phases are conceptually separate, and they are executed in sequence.

**Using feedback for architectural refinements: a thorough process:**

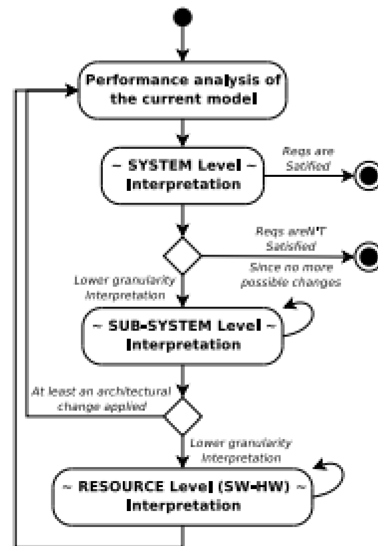


Figure 1: Results interpretation and feedback generation process

Figure 1 shows an activity diagram representing the main flow of the whole process for interpretation of results and feedback generation. The iterative nature of the process is obviously related to the progressive refinements that are brought on the system architecture while the interpretation of performance indices progresses. The refinement steps are driven by the suggestions defined in special data structures that we call interpretation matrices. One or more interpretation matrices are associated to each granularity level. In order to produce such suggestions the process also lays on the ability to recognize antipatterns in the architectural design.

A first performance model is built for the whole system. After results are obtained from the solution of the system-level performance model solution (i.e. topmost block in Figure 1), the first step consists in the interpretation of these results (i.e. SYSTEM level block in Figure 1). If all the requirements are satisfied then the process successfully stop without suggesting any change in the architecture. If some of the given requirements are not satisfied, then it is suggested to move to a lower granularity level that is, in this case, the subsystem level. The set of identified subsystems have to be sorted following a certain criterion that may depend on the application domain.

**IV. Supporting structures: some classified antipatterns**

A quite crucial role in the interpretation matrices is played by antipatterns. Indeed, almost always at the subsystem level (and sometimes at the resource level) the action to be taken for result interpretation and to find alternative scenarios consists of searching in the subsystem for an antipattern, that we define here below.

A design pattern is a standard solution for a known problem. An antipattern is in practice a negative pattern, in that it is a pattern whose presence into a design has negative effects that should be avoided. In our case we consider performance antipatterns [9] that produce effects on the system performance. For each known performance antipattern a refactoring mechanism can be provided to overcome it. The refactoring consists of a sequence of transformations, from the original architectural model to a target model, that improve system performance while preserving the system functionalities

Many antipatterns have been classified in literature [9, 10, 11]. In our work we have considered the ones that can feasibly applied, with appropriate tailoring, to software architectures for performance goals. In this section we provide evidence of two antipatterns. However, other classified antipatterns are available in [7].

The Blob antipattern reveals itself if a particular resource does the majority of the work in a software architecture while banishing the other ones to minor support roles. This situation is often easy to recognize looking at the performance results, because the "blobbing resource", that embeds many of the functionalities provided by the system, presents a very high utilization if compared to resources in its neighborhood. The left side of Figure 2 shows an example of such antipattern.

The density of lines within each resource indicates the intensity of the resource load. A poor distribution of the system intelligence evidently appears in Figure 2. In the right side of Figure 2 a refactoring has been made on the system by distributing the system logics over all the resources. A better performing pattern can be thus obtained. In the left side of Figure 3 the Unbalanced Extensive Processing antipattern is shown. It characterizes the scenario in which a specific class of requests generates a pattern of execution within the system that tends to overload a particular resource (or a set of resources). In other words the overloaded resource (i.e. typically the slowest one) will be executing a certain type of job very often, thus in practice damaging other classes of jobs that will experience very long waiting times and, in addition, leaving quite idle the following resources in the pattern. This scenario has negative effects on the mean response time of the whole system, especially for the requests that do not belong to the considered class, as well as on the whole system throughput.

The Unbalanced Extensive Processing antipattern can be recognized by observing the utilization of the resources along the pattern and the classes of jobs that they process. This antipattern can be refactored by introducing specific fast-paths for the service requests that do not overload the considered resource and/or that need a particularly fast service, as shown in the right side of Figure 3. Obviously the positive effects of this refactoring will be more pronounced for the requests that will use the fast-path, while the positive effects on the whole system depend on the percentage of this request type overall the served requests.

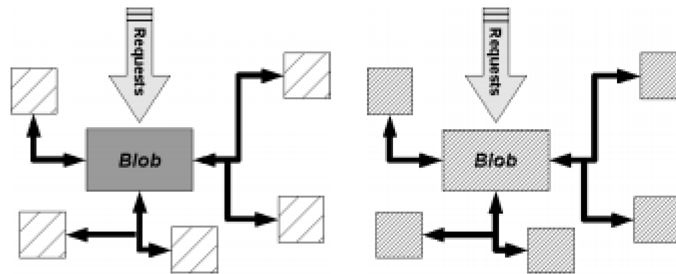


Figure 2: An example of Blob antipattern

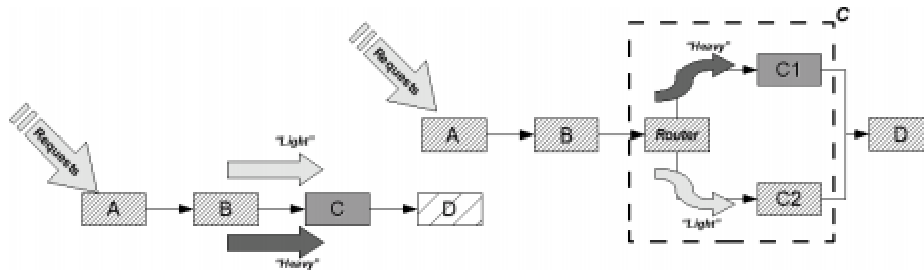


Figure 3: An example of unbalanced Extensive Processing antipattern

## V. CONCLUSION

In this paper, we have reviewed the state of the art in model-based software performance prediction. We have taken a software-designer perspective in order to classify and evaluate the growing number of approaches that have lately appeared. Our choice is driven by the generally acknowledged awareness that the lack of performance requirement validation in current software practice is mostly due to the knowledge gap between software engineers/architects and quality assurance experts rather than due to foundational issues.

## REFERENCES

**[1]** 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.

[2] R. Eramo, V. Cortellessa, A. Pierantonio, and M. Tucci, “Performance-driven architectural refactoring through bidirectional model transformations,” in QoSA, pp. 55–60, 2012.

[3] T. Mens and G. Taentzer, “Model-driven software refactoring,” in WRT, pp. 25–27, 2007.

[4] H. Harreld, “NASA Delays Satellite Launch After Finding Bugs in Software Program,” 1998.

[5] 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.

**[6**] S. Barber, T. Graser, J. Holt (2002) Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation, Proc. of the 17th IEEE ASE conference

[7] L. Frittella (2006) Feedback Architetturale Basato su Sistematica Interpretazione di Software Performance Analysis (in italian), Master Thesis, Università degli Studi dell’Aquila, Italy. http://www.di.univaq.it/cortelle/docs/TesiLaurento.pdf

[8] M. T. Su, Capturing exploration to improve software architecture documentation. Proc. 4th European Conference Software Architecture, Companion Volume, ACM Computing, ACM New York, 2010, pp. 17-21