A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis[⋆]

Vittorio Cortellessa and Laurento Frittella

Dipartimento di Informatica
Università dell'Aquila
Via Vetoio, 1, Coppito (AQ), 67010 Italy
cortelle@di.univaq.it,
laurento.frittella@gmail.com

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 that improve their architectural designs. The interpretation of results is based on the ability to automatically recognize performance anti-patterns in the software architecture. The whole process of result interpretation and generation of architectural alternatives is supported by a tool based on the Layered Queueing Network notation.

Keywords: Software Performance, Layered Queueing Networks, Architectural feedback, Performance indices.

1 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 drawback 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 [6]. In the last few years many efforts have been devoted to introduce automation in the

[⋆] This work has been partially supported by the PLASTIC project: Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication. EC - 6th Framework Programme. http://www.ist-plastic.org 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 Queueing 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 architectural 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.

Few related works can be found in literature that deal with the interpretation of performance results and the generation of architectural feedback. Most of them are based on monitoring techniques and therefore are conceived to only applied after software deployment for tuning its performance. We are instead interested to model-based approaches that can be applied all along the software lifecycle to support development decisions.

In [13] 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 [9], 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.

The Arcade tool, introduced in [2], is also based on a simulation model. Heuristic algorithms, in presence of detected system bottlenecks, are able to provide alternative solutions that practically remove the bottlenecks. The heuristics are based on architectural metrics that help to compare different solutions.

A quite interesting work has been introduced in [4], 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.

The paper is organized as follows: in Section 2 we illustrate our approach along with the structures and the entities that represent its core; in Section 3 we step-by-step apply our approach to a case study, and finally in Section 4 we provide conclusive remarks and future work.

2 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, in Section 2.3 we show how they need to share common knowledge on the system structure and its performance.

2.1 Software Performance Granularity: System, Subsystem, Resource

Software performance analysis can be conducted at different granularity levels. Indices like throughput, response time and utilization can be obtained from the performance analysis at the system level down to the single resource level.

A system can be logically split into several parts, and a detailed performance analysis restricted to the most critical partes can be conducted to better identify the adversary issues in a specific system’s area as soon as possible. Software architectures are by definition made of subsystems and components, therefore this "zooming" approach to the performance analysis finely applies to them.

In order to define a structural approach to the analysis of performance results, we have identified three granularity levels at which a software architecture can be analyzed, that are: System level, Subsystem level, Resource level.

System level - This is the highest abstraction level for conducting a performance analysis experiment; only global indices can be obtained by a system level analysis of the architecture, such as end-to-end response time (i.e. from the input to the output), system throughput, etc.

Subsystem level - This is an intermediate abstraction level where the system’s components and their interactions can be analyzed. In our approach this level does not have a fixed granularity, because any assembly of basic elements can be considered as a subsystem. We leave this definition as general as possible, so that the approach can be applied to multiple definitions of subsystems.

Zooming into architectural details (i.e. subsystem mechanism) can be driven by different strategy that aim at splitting the system following different criteria. Since our goal is to support the validation of a certain architecture vs a performance requirement, we devise two criteria for architecture splitting that depend on the type of performance requirement imposed on the system, as follows:

- *flat requirement*, i.e. one or more performance requirement are imposed on the whole system, no matter what is the service that the system will execute. An example of such requirement can be "The web server must be able to show a web page on the client side within 8 seconds from the request". In fact, this requirement must hold on the whole system, as it does not detail on the type of pages to show. To investigate such requirement, the system can be partitioned in subsystems that are clusters of components heavily coupled to perform a certain task. In this case the subsystems can be considered as *path-crossing* vs the path followed through the whole software architecture to satisfy a certain service request. In the remainder of the paper, the subsystems obtained with this type of splitting will belong to the *type1* category.
- *service oriented requirement*, i.e. one or more performance requirement are imposed on a specific system service. An example of such requirement can be "The web server must be able to show the catalog web page on the client side within 8 seconds from the request". This requirement holds only on a specific system service, that is a catalog request. To investigate such requirement, the system can be partitioned in subsystems such that each subsystem contains the components involved in a specific service provision. One of the major advantage in this type of splitting is that the performance requirements at the system-level can be easily associated with the subsystem that implements the service undergoing a requirement. The subsystems obtained with this type of splitting will belong to the *type2* category.

Note, however, that in both the above cases we do not exclude that two subsystems overlap each other, i.e. that a component can belong to more than one subsystem. This situation is more frequent in case of *type2* partitioning, as it will be seen later.

Resource level - It represents the finest grain level for conducting a performance analysis. Indices that can be obtained at this level are associated to a specific component. We assume here a general definition of component, that is: an atomic part of a system (software or hardware), that has an internal behavior and an external interface, and cannot be further split. At this level of granularity, the major difference between the two resource types resides in the changes that can be made on them to satisfy the performance constraints. For example, a hardware resource like a CPU can be duplicated to improve the throughput, whereas the duplication of a software component might improve the performance only if the two instances can be allocated on separate machines. For an overloaded software component, it is rather better to split the services that it provides among other unoccupied components.

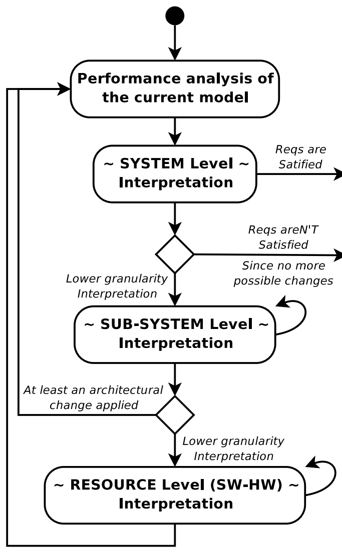

Performance analysis of the current model
~ SYSTEM Level ~ Interpretation
Reqs are Satified
Reqs areN'T Satisfied
Since no more possible changes
Lower granularity Interpretation
~ SUB-SYSTEM Level ~ Interpretation
At least an architectural change applied
Lower granularity Interpretation
~ RESOURCE Level (SW-HW) ~ Interpretation

Fig. 1. Results interpretation and feedback generation process

2.2 Using Feedback for Architectural Refinements: A Thorough 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* and that will be described in more details in Section 2.3. 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. The concept of antipattern within the performance domain and some examples of them are provided in Section 2.4.

Our assumption is that one or more performance requirements have been formulated for the whole system. If any requirements only refers to a specific portion of the system, then this process can be applied only to that portion by considering the latter as a whole system.

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[1]. The performance indices of the various subsystems are observed, and the focus is given to the worst one.

Subsystems are examined in a certain order (i.e. the loop on the subsystem level interpretation in Figure 1 represents this iteration) and if changes can be made without ambiguity on some subsystem (with the support of the interpretation matrices) then the process goes back to the first step and the updated performance model has to be solved. Otherwise, a further move to a lower granularity level is suggested: in this case, the resource level.

Analogous behavior of the process occurs at the resource level. After this interpretation step in any case the process brings back to the performance model solution to check whether the performance requirements are met or not.

2.3 The Interpretation Matrices

In our approach, the identification and construction phases share a structured knowledge about the system that we have organized in so-called *interpretation matrices*. Such matrices have a 2×2 format. The matrix rows represent interval of values for a certain performance index, and matrix columns do the same. In a (i, j) cell we describe the performance scenario that is characterized from the corresponding interval of indices values. If it is needed, we also define in a cell the actions that should be taken to find alternative scenarios.

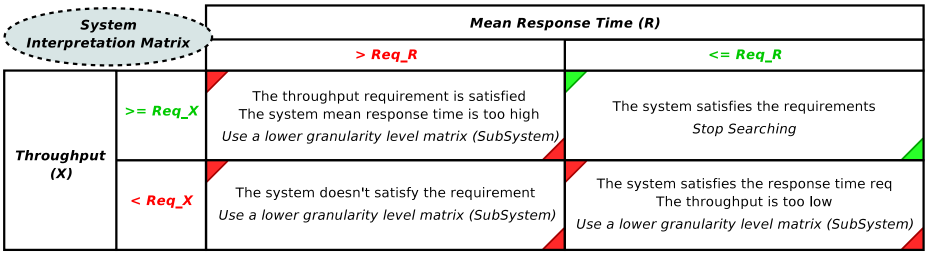
We have devised matrices for different levels of granularity, different splitting strategies of subsystems, and different types of components.

Figure 2 shows the matrix that we have built for system-level analysis (in [3] we present the other four matrices that we have defined). We assume that performance requirements at the system level must be formulated in terms of system throughput and response time ([2]). On the matrix rows we split the range of the system throughput in two intervals: the throughput values higher than the value Req_X specified in the requirement are associated to the upmost row of the matrix, whereas the lower values are associated to the bottommost row. On the matrix columns we represent the range of the system response time in two intervals: the values higher than the value Req_R specified in the requirement are associated to the leftmost column of the matrix, whereas the lower values are associated to the rightmost column.

In each cell of the matrix in Figure 2 we identify the performance scenario (in plain text), and we specify the next step (in italic text) to find an alternative scenario, if needed. For example, the lower leftmost cell represents the case of a low system-level throughput associated to a high response time. The matrix entry suggests to investigate at the subsystem level, and the designer has to choose one of the splitting strategies illustrated in Section 2.1. As opposite, the upper rightmost cell represents the case of a high system-level throughput associated to a low response time. The matrix entry

[1] The identification of subsystem is still a step that requires some human support, especially in case of flat requirement.

[2] We do not deal here with requirements on the utilization index, as it is quite rare to have such a requirement at the system and subsystem level. Utilization enters however in the picture at resource level of granularity.



System Interpretation Matrix		Mean Response Time (R)	
		> Req_R	<= Req_R
Throughput (X)	>= Req_X	🟥 The throughput requirement is satisfied The system mean response time is too high *Use a lower granularity level matrix (SubSystem)*	🟩 The system satisfies the requirements *Stop Searching*
	< Req_X	🟥 The system doesn't satisfy the requirement *Use a lower granularity level matrix (SubSystem)*	🟥 The system satisfies the response time req The throughput is too low *Use a lower granularity level matrix (SubSystem)*

Fig. 2. System Level Interpretation Matrix

suggests to stop the analysis because all the system requirements have been satisfied (recall that we assume all requirements at the system level).

2.4 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 [10] 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 [3].

Many antipatterns have been classified in literature [10,11,12]. 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 that will be used in the example provided in Section 3. However, other classified antipatterns are available in [8].

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 3 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 3. In the right side of Figure 3 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.

[3] In the remainder of the paper we will call software performance antipatterns simply as antipatterns, with few exceptions where differently specified.

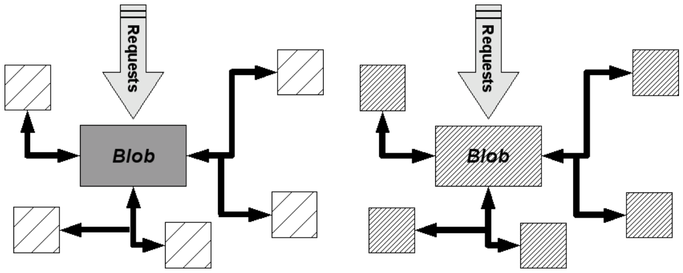


Fig. 3. An example of Blob antipattern

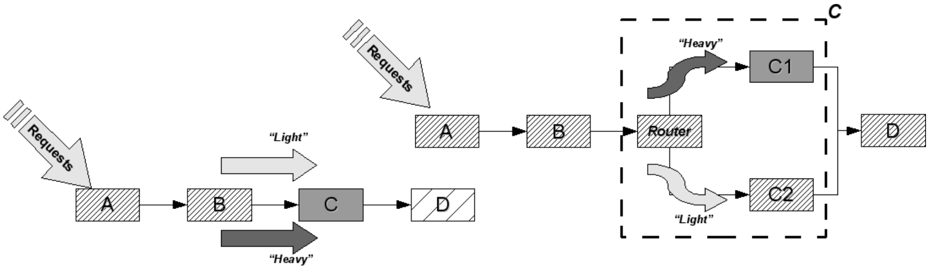


Fig. 4. An example of Unbalanced Extensive Processing antipattern

In the left side of Figure 4 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 4.

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.

3 Applying Our Approach

Our approach is not intended to be specific for a particular performance model, but for sake of experimental validation we need to choose a notation to instantiate the methodology and use it on a case of study. We have chosen the Layered Queued Networks [7,14].

The Layered Queuing Network (LQN) model is a canonical form for extended queueing networks with a layered structure. The layered structure arises from servers at one level making requests to servers at lower levels as a consequence of a request from a higher level. LQN was developed for modeling software systems, but it applies to any extended queueing network with multiple resource possession, in which multiple resources are held in a nested fashion.

The case study on which we have applied our approach represents a software architecture used for a small robot that can interact with the environment where it works and learns from its past experiences. The robot consists of three fundamental parts:

– *sensor machinery* - the robot makes use of sensors for visual perception, for measuring the environmental temperature and for communicating with other robots via wireless;
– *servosystems* - they enable the robot to move around and to interact, and possibly avoid, objects on its path;
– *computational engine* - this includes the intelligent and reactive components.

The main activity of the robot is to explore the whole environment around it and acquire knowledge for classifying events and sharing information with other robot-friends.

When an event happens either it is pointed out by the devices in the sensor machinery, or it is reported by one or more robot-friends that collaborate with the considered robot. The computational engine, using the acquired knowledge, establishes whether it is a potentially dangerous event or not and, in the latter case, it can be used to acquire new knowledge. The knowledge might also be acquired using the servosystems, for example by interacting with some objects on the ground. If an event is instead classified as dangerous, then the robot must quickly react by making suitable remarks and stopping itself before running into danger. An UML Sequence Diagram of a generic regular event handling is shown in [3].

We have modeled such system architecture in LQN, as shown in Figure 5. The *Environment* and *OtherROBOTS* tasks of Figure 5 are used only as request sources to generate the system workload and do not belong to the analyzed system.

Following the previous classification, the LQN tasks can be subdivided as:

– *sensor machinery*: Sensors, NetRX, NetTX;
– *servosystems*: Arms, Motors, MoveController;
– *computational engine*: StorageMemory, VolatileMemory, AI, Handler.

We assume that the number of sensors is fixed at 2 (i.e. visual and temperature input sensors) and the robot-friends number can instead vary from 1 up to 13. Besides, we defined the following performance requirements:

1. the robot must react in no more than 4.5 seconds from the moment in which an event is classified as dangerous;
2. the mean processing and reaction time for an event, from the moment in which it starts its path from the computational engine, must not exceed 11 seconds.

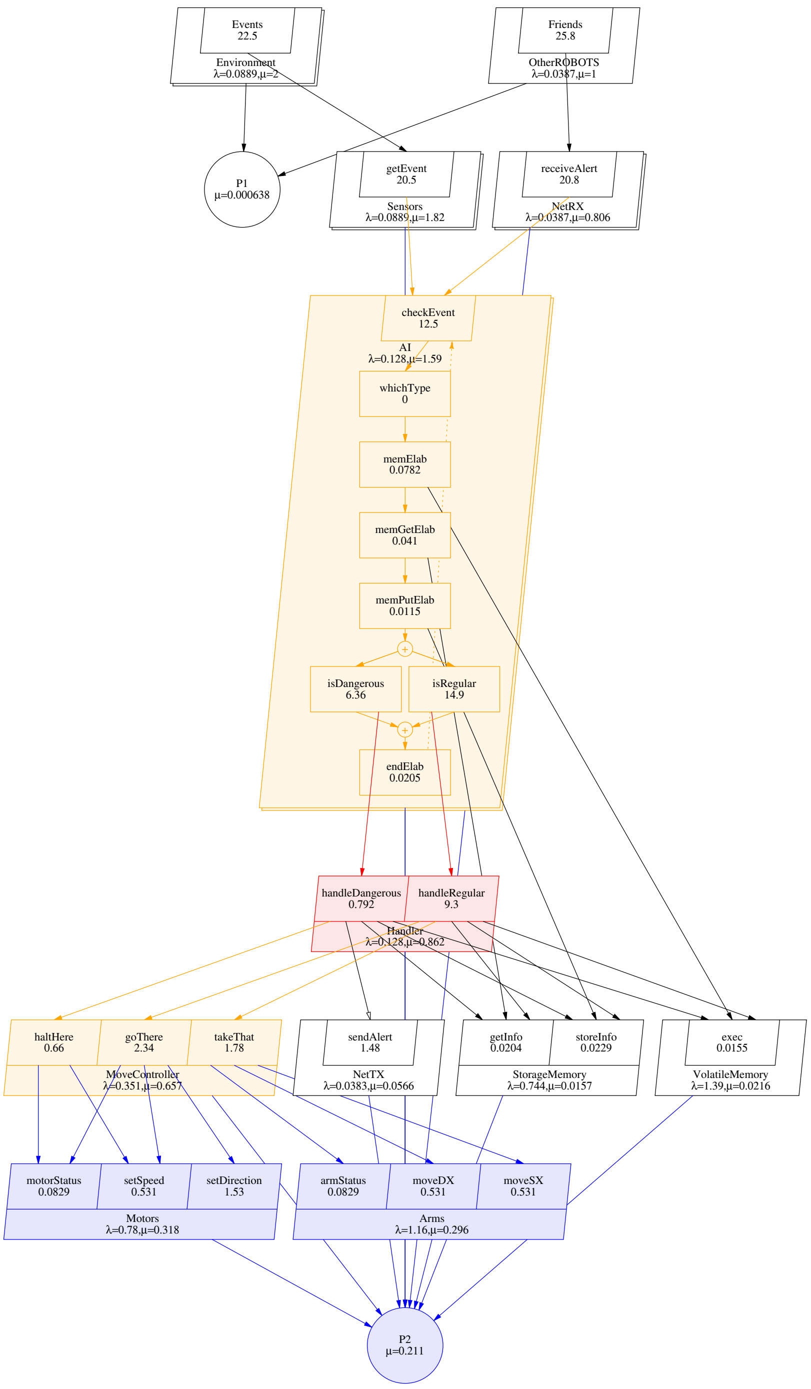


Fig. 5. The LQN model for the robot case study

Table 1. *Requirements and Performance Results - Iteration0*

	Target Value	*Current Value*
isDangerous (req1)	$\leq 4,5$	$\approx 9,25$
checkEvent (req2)	≤ 11	$\approx 15,5$

We associate the first requirement, in the LQN model, to the mean service time of the *isDangerous* entry. The second requirement is associated to the mean service time of the *checkEvent* entry. Both entries belong to the *AI* task.

In [3] all the parameters used for the initial LQN model are shown. The performance analysis of the initial model produces the results reported in Table 1. It is evident that the initial architecture does not satisfy the performance requirements. Our approach, basing on the system-level interpretation matrix of Figure 2, suggests to identify subsystems, in one of the previously described ways, for a finer grain performance analysis. In this case study we have both types of requirements, as classified in Section 2.1. The first one is service specific whereas the second one is related to the whole system (i.e. a flat requirement). At this point we have chosen to adopt a type2 system splitting, even though type1 could be used as well. Of course, depending on the type of splitting, the appropriate interpretation matrix has to be used in the next step.

SubS_dangerous is the first analyzed subsystem, and it is composed by all the system tasks with the exception of the *Arms* task.

The subsystem type2 interpretation matrix [3] has to be referred for actions to take.

In this case, high mean response time and a good throughput level ([4]) suggest to search for any known antipattern in the considered subsystem.

By observing the *Handler* task and the type of requests that run over the system, the “Unbalanced Extensive Processing” antipattern can be retrieved on it (see left side of Figure 6. In fact the considered task has a sufficiently high utilization level (i.e. about 86.2%) and it receives two different request types: one relates to the regular events processing (and consequently with potentially heavy environmental interactions), and the other one relates with the dangerous events which need a faster processing.

By applying the suggested solution to the retrieved antipattern, the refactored architecture, as shown in the right side of Figure 6, achieves the performance levels summarized in Table 2. Indices have been improved, the first requirement has been satisfied but the second one has still not been met.

The analysis should proceed with the goal of reducing the mean system response time for a generic event while considering that, in accordance with performance model parameters, the 70% of the captured events are classified as regular. Thus we will analyze the *SubS_regular* subsystem because its performance affects the global system performance more than the other subsystems.

In the new considered model the *Handler2* task belongs to the *SubS_dangerous* subsystem but the *Handler* task, that now does not offer any service for the dangerous events processing, only belongs to the *SubS_regular* subsystem, as shown in table 3.

[4] Note that no requirement has been imposed on the throughput, hence any value can be considered as feasible.

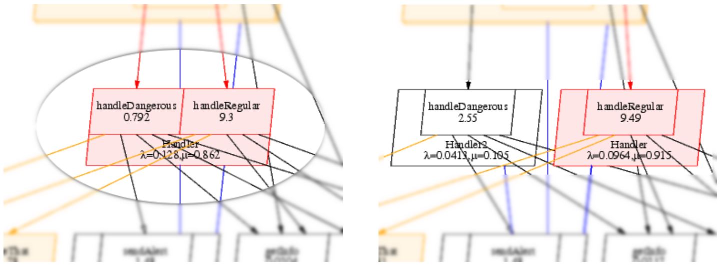


Fig. 6. Unbalanced Extensive Processing antipattern in robot system

Table 2. *Requirements and Performance Results - Iteration1*

	Target Value	*Current Value*	*distance from the previous iteration*
isDangerous (req1)	$\leq 4,5$	$\approx 3,3$	$-64,32\%$
checkEvent (req2)	≤ 11	$\approx 13,75$	$-11,29\%$

The subsystem type2 interpretation matrix used with the *SubS_regular* subsystem suggests to search for antipatterns in this case as well. Here the interactions among the tasks *MoveController*, *Motors* and *Arms* announce for a “Blob" antipattern, as shown in the left side of Figure 7.

The refactoring of the system due to the latter antipattern identification does not modify the model structure, but only the distribution of load, as shown in the right side of Figure 7.

This allows the software architecture to achieve the performance values summarized in Table 4.

The second requirement is still slightly over the desired level, so the analysis should make one more step. Now the subsystems do not contain any known antipattern, and the interpretation matrix suggests to go for a lower level of granularity and use the “software resource” interpretation matrix.

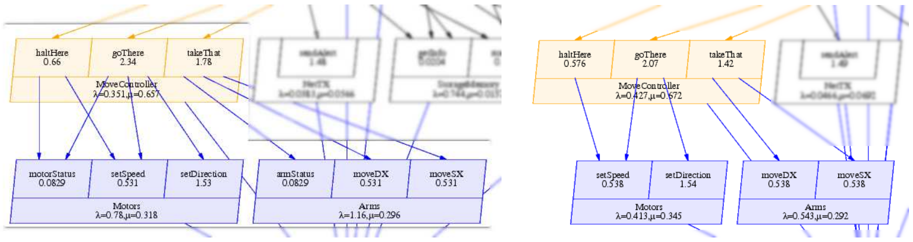


Fig. 7. Blob antipattern in robot system

Table 3. *SubSystems - Iteration2*

Software Resources	*SubSystem* ***SubS_dangerous***	*SubSystem* ***SubS_regular***
Sensors	$\surd$	$\surd$
NetRX	$\surd$	$\surd$
NetTX	$\surd$	
StorageMemory	$\surd$	$\surd$
VolatileMemory	$\surd$	$\surd$
AI	$\surd$	$\surd$
Handler		$\surd$
Handler2	$\surd$	
MoveController	$\surd$	$\surd$
Motors	$\surd$	$\surd$
Arms		$\surd$
SubSystem performance target	***R*** $\leq$ ***4,5*** *(on isDangerous)*	—
System performance target	***R*** $\leq$ ***11*** *(on checkEvent)*	

Table 4. *Requirements and Performance Results - Iteration2*

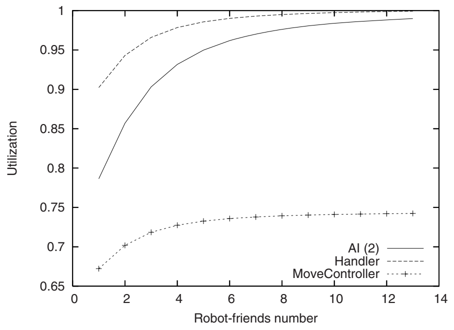
	Target Value	*Current Value*	*distance from the previous iteration*
isDangerous (req1)	$\leq 4,5$	$\approx 3,15$	$-4,54\%$
checkEvent (req2)	≤ 11	≈ 12	$-12,73\%$

The first analysis consists of examining the utilization level for the resources belonging to the considered subsystem to find the one in the worst state. As shown in the left side of Figure 8, the *Handler* resource has the highest utilization value and the “software resource” interpretation matrix suggests to clone it. Thus we have raised its resource multiplicity in the LQN model.

This change has positive effects on the generic event processing performance although it is not enough to satisfy the requirements. Thus we can consider the *AI* resource that is the current most used resource in the *SubS_regular* subsystem, as shown in the right side of Figure 8. However, the raise of its multiplicity has negative effects, very likely because the number of requests in the queues of other system resources becomes too high. For this reason, we did not apply this change.

Handler is the second highly used resource in the subsystem and its utilization level is over 80%, as shown in the right side of Figure 8. Raising its multiplicity, as suggested by the proper interpretation matrix, is in this case useless because the performance levels remain unchanged, so we did not apply this change either.

At this point, considering that the hardware (like CPU and memories) which is directly used by the software components can support the current workload with medium utilization levels, we can try to improve the hardware related with the servosystems, i.e. the *Motors* and *Arms* tasks that are the slowest components of the whole robot system.



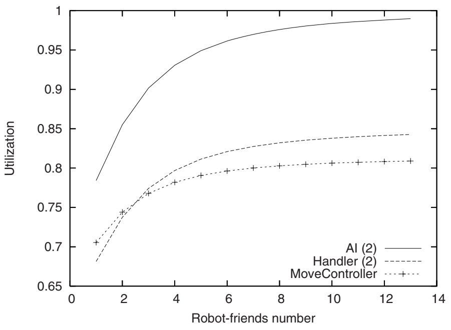


Fig. 8. Resource Utilization Graphs

Table 5. *Requirements Summary - Iteration3*

	Target Value	*Current Value*	*distance from the previous iteration*
isDangerous (req1)	$\leq 4,5$	≈ 4	$+26,98\%$
checkEvent (req2)	≤ 11	$\approx 10,3$	$-14,17\%$

Thus, basing on the hardware resource interpretation matrix, we decided to drop the delay of each servosystems activity by 0.1 seconds. This leads to a considerable performance increase. In fact, at the end of the process the performance goals are achieved, as shown in table 5.

4 Conclusions

We have presented an approach to interpret performance analysis results and generate architectural feedback on the basis of result interpretation. Using our approach, guidelines for interpretation and a thorough process can be followed to break the adversary design choices that negatively affect the system performance.

Although we have implemented a prototyped tool that may guide the developers along the whole process, it is still necessary some human experience in several steps. For example, the detection of antipatterns in a subsystem is a task whose complexity heavily depends on the structure of the subsystem and the definition of the antipattern itself. However, at the best of our knowledge, this is the first work that embeds in the same process the interpretation of performance results and the formulation of architectural alternatives. In addition, we have given a first (still preliminary) contribution to structuring the knowledge necessary for such task.

As future work we mainly intend to consolidate the antipattern definitions and retrieving. Consequently, we can improve the tool support to the whole process. Interpretation matrices are still too informal, thus more effort shall be dedicated to their refinement in order to apply our approach to more complex case studies. Besides, we plan to extend our approach by considering possible architectural constraints that may prevent from applying suggested changes (e.g. a certain load cannot be distributed on other components due to "narrow" connectors). The whole approach does not depend on the notation adopted to represent the performance model, however it will be interesting to experiment on other notations such as Petri Nets. As a long-term goal, we plan to introduce cost issues in the choice of architectural alternatives, exactly like CBAM process suggests [5].

References

1. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based Performance Prediction in Software Development: A Survey. IEEE Trans. on Soft. Eng. 30(5), 295–331 (2004)
2. Barber, S., Graser, T., Holt, J.: Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation. In: Proc. of the 17th IEEE ASE conference. IEEE Computer Society Press, Los Alamitos (2002)
3. Cortellessa, V., Frittella, L.: A framework for automated generation of architectural feedback from software performance analysis, TRCS 007-2007, Technical Report, Dipartimento di Informatica, University of L'Aquila (2007), http://www.di.univaq.it/cortelle/docs/feedbackreport.pdf
4. Dobrzanski, L., Kuzniarz, L.: An Approach to Refactoring of Executable UML Models. In: Proc. of ACM SAC. ACM Press, New York (2006)
5. Kazman, R., et al.: Quantifying the Costs and Benefits of Architectural Decisions. In: Proc. of ICSE01 (2001)
6. Lazowska, E., et al.: Quantitative System Performance - Computer System Analysis Using Queueing Network Models. Prentice-Hall Inc., Englewood Cliffs (1984)
7. Franks, G., et al.: Layered Queueing Network Solver and Simulator User Manual. Tech. Report, Department of Systems and Computer Engineering, Carleton University (2005), http://www.sce.carleton.ca/rads
8. Frittella, L.: Feedback Architetturale Basato su Sistematica Interpretazione di Software Performance Analysis (in italian). Master Thesis, Universitá degli Studi dell'Aquila, Italy (2006), http://www.di.univaq.it/cortelle/docs/TesiLaurento.pdf
9. Sancho, P., Juiz, C., Puigjaner, R.: Automatic Performance Evaluation and Feedback for MASCOT designs. In: Proc. of the 5th ACM WOSP. ACM Press, New York (2005)
10. Smith, C., Williams, L.: Software Performance AntiPatterns. In: Proc. of 2nd ACM WOSP. ACM Press, New York (2000)
11. Smith, C., Williams, L.: New Software Performance AntiPatterns: More Way to Shoot Yourself in the Foot. In: Proc. of CMG international conference (2002)
12. Smith, C., Williams, L.: More New Software Performance AntiPatterns: Even More Ways to Shoot Yourself in the Foot. In: Proc. of CMG international conference (2003)
13. Williams, L., Smith, C.: PASA: An Architectural Approach to Fixing Software Performance Problems. In: Proc. of CMG international conference (2002)
14. Woodside, M., Franks, G.: Tutorial Introduction to Layered Modeling of Software Performance, Tech. Report, Department of Systems and Computer Engineering, Carleton University (2005), http://www.sce.carleton.ca/rads