Software Performance Model-Driven Architecture

Vittorio Cortellessa
Dipartimento di Informatica
Universita' dell'Aquila
Via Vetoio, Coppito (AQ), Italy
cortelle@di.univaq.it

Antinisca Di Marco[*]
Computer Sc. Department
University College of London
Gower Street, London, UK
a.dimarco@cs.ucl.ac.uk

Paola Inverardi
Dipartimento di Informatica
Università dell'Aquila
Via Vetoio, Coppito (AQ), Italy
inverard@di.univaq.it

ABSTRACT

Model transformations in MDA mostly aim at stepping from a Platform Independent Model (PIM) to a Platform Specific Model (PSM) from a functional viewpoint. In order to develop high quality software products, non-functional attributes (such as performance) must be taken into account. In this paper we extend the canonical view of the MDA approach to embed additional types of models that allow to structure a Model Driven approach keeping into account performance issues. We define the relationships between MDA typical models and the newly introduced models, as well as relationships among the latter ones. In this extended framework new types of model-to-model transformations also need to be devised. We place an existing methodology for transforming software models into performance models within the scope of this framework.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; C.4 [**Computer Systems Organization**]: Performance of Systems; I.6.5 [**Simulation and Modeling**]: Model Development

Keywords

Software Performance, Model Driven Engineering

1. INTRODUCTION

Model Driven Architecture (MDA) [8] has given the opportunity to software scientists and developers to converge towards a common framework that defines the theory and practice of model-driven engineering. The typical 3-steps structure of MDA (i.e. CIM, PIM, PSM) is proving to be a good abstraction of many development practices in model-based software engineering. Indeed, independently of the development process, it is undeniable that a model of requirements must be built at the beginning of the process (Computation Independent Model), then it inputs the architectural design phase that produces a model of the system logics (Platform Independent Model), and finally the latter model inputs the implementation phase that produces the system code for a certain platform (Platform Specific Model).

Although MDA nicely represents the *functional* aspects of a model-driven software development process, it falls short to represent the modeling and analysis necessary to guarantee *non-functional* properties of software products. In modern distributed software systems, often based on mobility of devices and code, non-functional attributes, such as performance and availability, become first-class entities. Indeed the misbehavior of such systems on field is often given by their inability to meet non-functional requirements.

A trivial example is given from the famous experimented 8-seconds rule in a web page access: it is mandatory for a web application to be designed so that it takes no more than 8 seconds to completely download a page on a client site, otherwise the user migrates to another site with an obvious loss of opportunity for the site owners.

Hence the integration of non-functional modeling activities, along the whole software lifecycle, is nowadays necessary to take into account these properties from the beginning of the software development process and to deliver software products that meet the requirements.

In the last few years the research in software performance validation has faced the challenge of automated generation of performance models from software artifacts. Several methodologies have been introduced that share the idea of annotating software models with data related to performance and then translating the annotated model into a performance model ready to be validated.

Goal of this paper is to embed the performance validation activity in the MDA framework. We propose an extension of the MDA framework, namely SPMDA (Software Performance Model Driven Architecture) that, beside the typical MDA models and transformations, embeds new models and transformations that take into account the performance validation activities. To this goal, we need to define new types of model transformations, that are different from those in MDA whose aim is to transform less refined models in more refined ones up to the automated code generation. In a software performance analysis context, instead, we interpret model transformation as an activity that does not necessarily aim at producing more refined models, but also to allow

[*]*Current address*: Dipartimento di Informatica, Universita' dell'Aquila, Via Vetoio, Coppito (AQ), Italy, adimarco@di.univaq.it.

a change of model notation that may favor different types of analysis. Hence, in SPMDA we introduce the concept of horizontal transformation, that is a transformation between models at the same level of detail looking at the system from different viewpoints. For example, a horizontal transformation can be devised from a set of UML diagrams representing a software architecture annotated with performance data to a Queueing Network representing the same software architecture in a different formalism ready to be analyzed.

This work lays on our knowledge of a large number of methodologies to transform software models into performance models, basing on which we have extrapolated the main methodological aspects that these approaches share, and we aim at giving an unifying vision of the domain.

About ten years ago the software performance validation discipline has lived an actual breakthrough due to the introduction of techniques based on model transformations [14]. These new techniques aimed at integrating this activity in the software lifecycle without changing the daily practices of software developers. Very successful methodologies have been created that allow to annotate software models (in different notations that span from ADLs to Process Algebras to UML) with performance data (such as the operational profile), and allow to transparently translate the annotated software model into a performance model (e.g., a Queueing Network or a Petri Net). A recent comprehensive survey on methodologies and tools for model-to-model transformations in software performance can be found in [1]. None of these methodologies explicitly defines the roles of its models within an MDA context.

Within MDA, a very interesting viewpoint has been introduced in [10]. The authors aim at helping designers to reason on non-functional properties at different levels of abstraction, likewise MDA does for functional aspects. They introduce a development process where designers can define, use and refine the measurement necessary to take into account QoS attributes. The work in [10] differs from ours because their ultimate goal is either to generate code for runtime monitoring of QoS parameters, or provide a base for QoS contract negotiation and resource reservation in the running system.

In [12], MDA is viewed as a suitable framework to incorporate various analysis techniques into the development process of distributed systems. In particular, the work focuses on response time prediction of EJB applications by defining a domain model and a mapping of the domain onto a Queueing Network metamodel. This work can be considered as another instance of our framework, much easier to integrate than other ones due to the effort spent from authors to match the MDA concepts.

This paper is organized as follows: in section 2 we describe our framework (i.e. the new models and transformations), in section 3 we instantiate the framework on a software performance validation approach, and finally in section 4 we give the conclusive remarks and the future implications of our work.

2. SOFTWARE PERFORMANCE IN MDA

In this section we present our extended view of the MDA approach illustrated in Figure 1. On the left side of the figure the canonical MDA models and transformations appear: a Computation Independent Model (CIM) becomes a Platform Independent Model (PIM) upon specifying business logics of the system; PIM becomes a Platform Specific Model (PSM) once the business logics is implemented on a particular platform [8].

As said in section 1, these models and transformations do not suffice to keep performance aspects under control during the software system development. In order to address this issue, in Figure 1 three additional types of models and three additional types of transformations/relationships have been introduced. The new types of models are: CIPM, PIPM, PSPM.

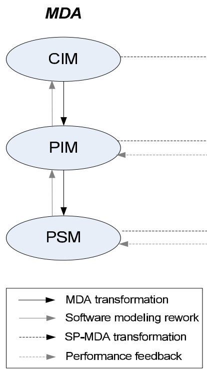

Figure 1: The SPMDA framework.

CIPM - A Computation Independent Performance Model represents the requirements and constraints related to the performance. Very often the performance requirements concern the response time of some system operations under certain workloads (e.g. *"the authentication service must be completed in average within 2 seconds under a workload of maximum 100 users"*). In the MDA framework the UML Use Case Diagrams are the most widely used diagrams to model system requirements and their relationships. Therefore a good example of CIPM may be a Use Case Diagram annotated with performance requirements. In Figure 2 we represent the requirement formulated above.

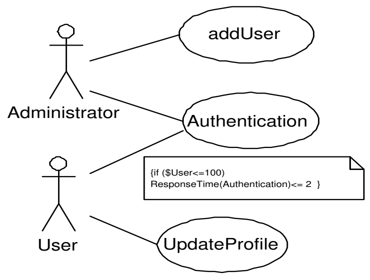

Figure 2: An example of CIPM.

PIPM - A Platform Independent Performance Model is a representation of the business logics of the system along with an estimate of the amount of resources that such logics needs to be executed. The model must be computationally solvable, that means it should be expressed through a notation that allows performance analysis (e.g., Queueing Networks) ([1]). A platform independent performance analysis cannot conform the canonical interpretation aimed at

[1]An UML model annotated with performance data is not

obtaining numerical values of performance indices. This is due to the tight relation between typical performance indices (such as response time) and platform characteristics (such as network latency) that are not available in a PIPM. Within the PIPM class we rather intend to represent models that return "a certain measure" of performance that allows to avoid early design bugs. At this level of detail, given a set of resource types, one can couple to any action of the business logics an estimated amount of each resource type needed to execute the action. For example, the amount of resources needed to execute the logics implementing an authentication service might be as follows: N high-level statements, K database accesses and M remote interactions. Being the estimates expressed in non-canonical units of measure, the results of the analysis of a PIPM cannot be used as a target for comparison to the actual system performance. They are rather meant: (i) to evaluate upper and lower bound in the system performance, (ii) to identify software performance bottleneck in the business logics (e.g. a software component that is overloaded), (iii) to compare the performance of different design alternatives of the business logics (e.g. two different software architectures).

PSPM - A Platform Specific Performance Model contains the merged representation of the business logics and the platform adopted to run the logics. In a classical MDA approach a platform is represented from a set of subsystems and technologies that provide a coherent set of functionalities through interfaces and specified usage patterns (e.g. J2EE, CORBA, etc.). In a performance context a platform must also include the characteristics of the underlying hardware architecture, such as the CPU speed and the network latency. Being expressed in classical units of measures (e.g. execution time), the results of the analysis of a PSPM can be used as a target for comparison to the actual system performance in order to validate the model. Such a model can then be used to explore the system behavior in scenarios inaccessible in the real world (e.g. extremely heavy workloads).

Behind PIPM/PSPM duality there is the intuitive concept that the performance analysis results can be expressed with actual time-based metrics only after a PIM is bounded to its platform and becomes a PSM.

Obviously the results coming out a PIPM evaluation are not useful to validate the model against the system requirements, because they may take very different time values on different platforms. However, three types of actions can originate from this analysis:

- **Lower and upper bounds** on the system performance can be evaluated if some estimates of the performance of the possible target platforms are available. For example, if the lower bound on a system response time is larger than the corresponding performance requirement, then it is useless to progress in the development process as performance problems are intrinsic in the software architecture. It is necessary to rework on the software models. However, even when the results are not so pessimistic it is possible to take decisions that improve the software architecture.

- In order to identify the most **overloaded components**, the utilization and/or queue length of each service center in our PIPM must be computed vs the system population. An overloaded component has a very long waiting queue and represents a bottleneck in the software architecture. Some rework is necessary in the PIM to remove the bottleneck.

- Either as a consequence of the above decisions or as a planned performance test, different (functionally equivalent) **alternative software designs** can be modeled as PIMs, and then their performance can be compared through their PIPMs in order to select the optimal one.

a PIPM because it cannot be solved to obtain performance indices, while a Queueing Network does.

However, all the above evaluations are meaningful only if the estimates of the model parameters are reliable. This is an issue that may be raised for any type of model at any level of detail. Moreover, in the early software development phases, there is often lack of knowledge of performance aspects, and trustable parameter estimations are even harder to achieve. On the other end, accurate techniques have been introduced to estimate performance model parameters. For example, the operational profile can be estimated by exposing a rapid prototype of the system to sample users and collecting the usage profile [13].

In order to complete the description of our framework, let us go back to the Figure 1 and illustrate the transformations and relationships that we have introduced among the models. The continuous arrows with single filled peak represent the typical MDA transformations that allow to build a PIM from a CIM and a PSM from a PIM. The dashed arrows with single filled peak represent the additional transformations that we have introduced to tie MDA and SPMDA models. We distinguish between *horizontal* and *vertical* SPMDA transformations, as follows:

SPMDA horizontal transformation - It transforms a software model into the corresponding performance model at any level in the MDA hierarchy. In Figure 1 CIM→CIPM, PIM→PIPM and PSM→PSPM are horizontal transformations. The model transformations belonging to this class share a two-steps structure: the software model is first annotated with data related to the system performance (e.g. the operational profile), thereafter the annotated model is transformed into a performance model. Many examples of such transformation have been introduced in the last few years, as referred in section 1.

SPMDA vertical transformation - Even though it seems to play in the performance domain a similar role to the one of MDA transformations, such transformation is instead intended to provide an input contribution to the horizontal transformation. In other words, often the horizontal transformations needs some input from the one-step-higher performance model in the hierarchy, hence: CIPM→PIPM is a contribution to a PIM→PIPM transformation and, likewise, PIPM→PSPM is a contribution to a PSM→PSPM transformation.

In section 3 we instantiate the SPMDA framework on a thorough approach that works at platform independent and platform dependent level.

Two types of arrows with double empty peak also appear in Figure 1 to give completeness to the software performance analysis process, and they represent the reverse paths after the performance analysis takes place.

Dashed arrows with double empty peak represent the feedback that originates from the evaluation of a performance model. The feedback can be expressed in terms of possible changes to apply to the software model in order to solve possible problems emerged in the software design upon the performance analysis. In Figure 1 there is no feedback only from CIPM to CIM, because, as we will show in section 3, CIPM is almost always a non-solvable model.

Continuous arrows with double empty peak are direct consequences of performance feedback. They represent the rework necessary on the software models to embed the changes suggested from the performance analysis. Indeed, the performance feedback may report so serious problems in the software modeling side that it does not suffice to work at the same level of detail to solve the problem. And, analogously to what happens sometimes after a software testing phase, the problem must be reported to previous phases in the development process to be solved. For example, the solution of a PSPM may suggest that a software component is overloaded and represents a performance bottleneck. This problem can be solved at PSM level if, for example, a change in the software-to-hardware mapping is sufficient to remove the problem. Viceversa, a software modeling rework is necessary at PIM level because the component needs, for example, to be split in more than one. Still, if this is not feasible due to some constraint on that component, the problem has to be reported at the requirement level, and some rework is necessary on the CIM.

The main advantage of horizontal transformations, as intended within the scope of this paper, is to enlarge the goals of model transformations beyond the classical ones, that are: refining of models towards code generation, computing and synchronizing different model views. The target models of horizontal transformations are always different from the source ones (with few exceptions), and are generated to be solvable and to provide non-functional indices of the source model. Although few efforts have been made to formalize some methodologies in the graph transformation domain, a major problem is that there is still lack of formal rules for horizontal transformations. As a consequence, it is not clear whether the existing languages are powerful enough to express them or specific languages are needed.

3. AN SPMDA FRAMEWORK INSTANCE

In this section we embed within the SPMDA framework an existing approach to transform software models into performance models, namely **Sap·One** [4].

The **Sap·One** methodology has been introduced in [4] and has been used in [5] for performance analysis of distributed systems. It was originally conceived as a model-based approach to estimate the performance of software architectures. The prerequisites to apply the approach consist of modeling the software systems by means of UML 2 diagrams, as follows: modeling the system requirements through an Use Case Diagram, modeling the software architecture through Component and Sequence Diagrams, and modeling the software-to-hardware mapping through an Deployment Diagram.

Figure 3 represents the **Sap·One** approach within the SPMDA framework.

Let us first describe the software side of the figure, i.e. the software models that represent the MDA process. The system requirements are expressed by an Use Case Diagram that represents the Computation Independent Model of the system. The Platform Independent Model consists of a software architecture expressed by: (i) a Component Diagram that models the static aspects, such as the offered/required services of the components and the architecture topology; (ii) a set of Sequence Diagrams (one for each Use Case) that model the application dynamics through a set of system scenarios. Finally, the Platform Specific Model contains different types of information: the name of the platform (e.g. CORBA, J2EE, etc.) determining the type of code that is generated, and a Deployment Diagram determining the software-to-hardware mapping of the application.

In the right-hand side of Figure 3 the additional models introduced in our framework are represented.

The topmost model, i.e. the Computation Independent Performance Model, is obtained by the CIM with notes attached to use cases to express the performance requirements of the system. Note that the CIPM can contain a subset of the CIM since in the CIPM only system requirements subject to performance constraints are reported. These constraints can be expressed in any unambiguous language, and they represent the requirements that our integrated framework aims at keeping into account during the whole development process. As an example, in Figure 3 a constraint on the response time of *Service1* requires that the mean response time of such service must not be larger than 1 second when the customers in the system are less than 200. A similar constraint is associated to *Service2*.

The Platform Independent Performance Model is represented by a multi-chain Queueing Network model (QN) [7]. The adopted semantics of the QN associates one software component to each service center, and the QN customers represent the different requests that users make to the software system.

As we have remarked in section 2, each SPMDA transformation is a 2-steps process: model annotation and annotated model transformation. In this case, the PIM is annotated with the following data: (i) Component Diagram - service demand of each component service, scheduling policy of each component on pending service requests; (ii) Sequence Diagram - workload description, probabilities over the branching points, average number of loop iterations.

In Figure 4 we show an example of annotated PIM, where the annotations come from the UML profile for Schedulability, Performance and Time (SPT) [11].

In the Component Diagram we annotate each component by means of the `<<PAhost>>` stereotype. Indeed, **Sap·One** considers software components as (abstract) active resources able to process tasks. Through such stereotype, the annotation indicates the scheduling policy of the software components (note that in the figure all components have *FIFO* scheduling policy). Each provided service is, instead, annotated with `<<PAstep>>` stereotype to indicate the service demand (i.e., `PAdemand` tag value) that the service requires to the component to be accomplished. The service demand is expressed in number of *high-level operations*, that represent a measure of the complexity of the steps the component has to execute to provide the required service. For example, *op1* service provided by *Comp2* has associated a service demand equal to 2 *high-level operations*. This means that *Comp2* executes 2 *high-level operations* to provide *op1*.

The same `<<PAstep>>` stereotype is also used in the Sequence Diagram to annotate probabilities in the alternative system behaviors (i.e. `PAprob` tag value) under the constraint that the sum of the probabilities of all alternatives must be equal to 1. In the Sequence Diagram of Figure 4

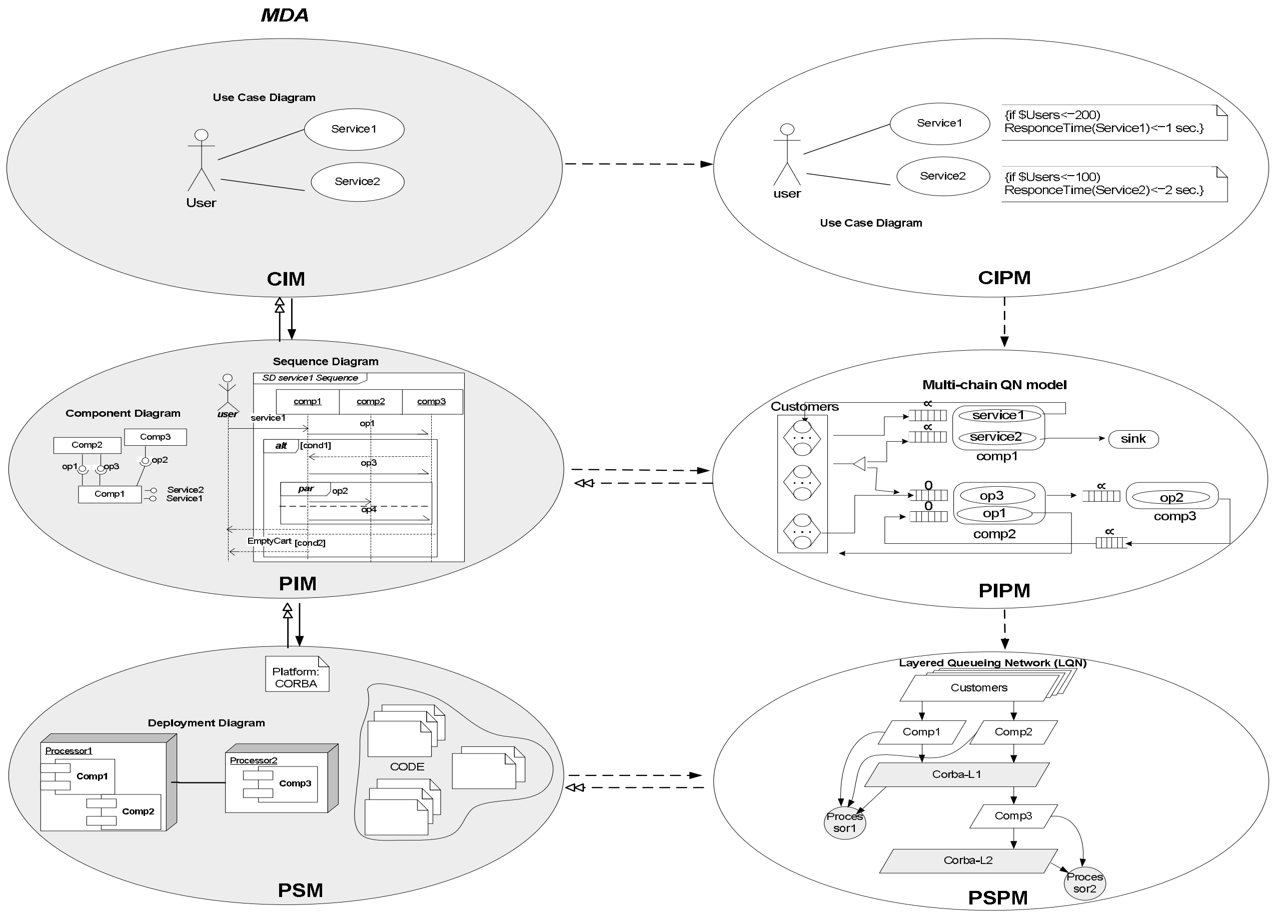

Figure 3: The Sap·One approach.

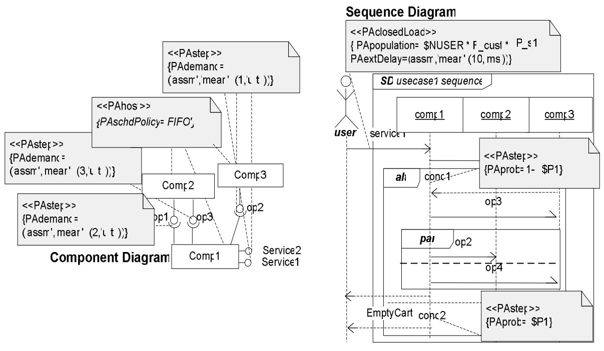

Figure 4: Examples of Sap·One annotations.

there are two alternatives annotated with parametric probabilities (i.e. `$P1` and `1-$P1` respectively) whose sum is always 1. Finally, the Sequence Diagram is also annotated with the `<<PAclosedLoad>>` stereotype that describes the arrival process of the requests.

The annotated PIM (i.e. Component Diagram plus Sequence Diagrams) is then transformed into a multichain QN as follows. Each component becomes a service center of the network, and its characteristics are extracted from the annotations illustrated above. The QN topology reflects the one of the Component Diagram. Each Sequence Diagram is processed to lump into a class of jobs (i.e. a chain) of the QN. A job traverses the network following the behavior described by the Sequence Diagram it comes from. The network workload is extracted from the annotations in the Sequence Diagrams [3]. In this step, the CIPM brings the target performance results to be compared with the one obtained from the PIPM solution.

The semantics of the QN model built in this step gives evidence to the inability of pursuing a "punctual" performance assessment at this level in the MDA hierarchy. The software annotations are expressed in high-level units of measure, likewise the results of the analysis will be. However (as we discussed in section 2) lower and upper bound analysis, software bottleneck identification and comparison of alternative software architectures can be carried out basing on this analysis results.

Going down in the hierarchy of Figure 3, the Platform Specific Model can be transformed into a PSPM. Layered Queueing Networks (LQN) [9] are adopted in **Sap·One** as a PSPM. An LQN is a collection of software tasks and hardware resources, with a layered interconnection topology. Higher layer tasks are clients of lower level ones, and lower level tasks are servers of the higher level ones. A task can be at the same time client of some tasks and server of other ones. For example, in Figure 3 *Comp1* is client of *Corba-L1* task and server of *Customers* task. Tasks in the topmost layer represent system requests made by the users, whereas round nodes in the bottommost layers represent hardware resources. Beyond the graphical notation, the LQN solver tool [6] builds a QN for each pair of adjacent layers of the model. The model solution iteratively solves the QNs starting from the two bottommost layers and going up in the LQN.

In the generation of the PSPM the approach uses the PIPM as follows: each service center in the QN model (PIPM) becomes a task in the LQN model (PSPM). The PSPM is enriched by: (i) new tasks modeling the MDA platform chosen to generate the PSM (e.g. CORBA) that are placed appropriately by visiting the code; (ii) LQN devices whose description is extracted from the Deployment Diagram available in the PSM. For example, in Figure 3 the LQN model is enriched by two new tasks that model the CORBA technology (i.e. *Corba-L1* and *Corba-L2*) and two devices modeling the nodes in the Deployment Diagram (i.e. *Processor1* and *Processor2*).

At this level in the MDA hierarchy a different type of performance analysis is pursuable. The LQN that we build embeds all the software and hardware parameters that are needed for a canonical performance analysis. They have been collected and/or monitored on the actual deployed system. End-to-end response time, utilization and throughput of any platform device can be computed by solving the LQN model, and the results can be compared to the measures on the actual system. Once validated, such model can be used for performance prediction. System configurations and workload domains that are unfeasible to explore in the actual system can be represented in the model in order to study the behavior of the system under different performance scenarios.

The outputs of the PSPM evaluation represent the feedback on the PSM, which needs to be modified if performance do not satisfy the requirements. As illustrated in section 2, the model rework on the software side could propagate up to the higher level models in case no feasible change can be made on the PSM to overcome the emerging performance problems.

4. CONCLUSIONS

The necessity of integrating the performance validation task along the whole software lifecycle, due to the performance criticality of modern software systems, is the rationale behind this paper. We have built a structure (SPMDA) around the Model-Driven Architecture framework in order to keep into account the models and the transformations related to the performance validation activity.

Performance models with analogous roles of CIM, PIM and PSM have been devised (namely CIPM, PIPM and PSPM) and new types of model transformations and relationships have been introduced to tie these performance models to the ones in the MDA hierarchy. In particular, we have introduced horizontal transformations that allow to transform a model at a certain level of detail into another, at the same level of detail, that gives a different view of the underlying system.

As an example, we have illustrated how an existing approach to the performance model validation can be plugged into our framework. Basing on our experience, many other approaches match this framework structure, that may represent an unifying vision of performance validation in MDA.

With regard to the performance validation domain, a very interesting challenge in the near future is to introduce automation in the feedback path, which transforms performance results into design alternatives.

From a wider angle, as a future perspective we retain that similar frameworks can be built for different types of non-functional attributes, such as reliability. Platform independent and platform dependent concepts belong to many software validation processes, and bringing them to the evidence of an MDA structure is very advantageous to make these activities acceptable from the software engineering community.

Finally, working on a classification of such types of model transformations, for example within the framework proposed in [2], would help to introduce formal tools in this domain and to answer to the question on appropriate languages that we have posed at the end of section 2.

5. REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-based Performance Prediction in Software Development: A Survey, *IEEE Trans. on Software Engineering*, 30(5):295-310, 2004.

[2] K. Czarnecki, S. Helsen, Classification of Model Transformation Approaches, *Proc. of OOPSLA'03, Workshop on Generative Techniques in MDA*, 2005.

[3] A. Di Marco, Model-based performance analysis of software architectures, Ph.D. thesis, University of L'Aquila, 2005.

[4] A. Di Marco, P. Inverardi, Compositional Generation of Software Architecture Performance QN Models, *Proc. of WICSA 2004*, pp. 37-46, 2004.

[5] A. Di Marco, F. Lo Presti, Two Early Performance Analysis Approaches at work on Simplicity System, *Proc. of QoSA, NetObjectDays 2005*, 2005.

[6] G. Franks et al., A toolset for performance engineering and software design of client-server systems, *Performance Evaluation*, 24(1-2):117-135, 1995.

[7] E.D. Lazowska et al., Quantitative System Performance: Computer System Analysis Using Queueing Network Models, Prentice-Hall, 1984.

[8] J. Miller (editor), Model-Driven Architecture Guide, omg/2003-06-01 (2003).

[9] J.A. Rolia, K.C. Sevcik, The method of layers, *IEEE Trans. on Software Engineering*, 21(8):689-700, 1995.

[10] S. Rottger, S. Zschaler, Model-driven development for non-functional properties: refinement through model transformation, LNCS 3273, pp.275-289, 2004.

[11] B. Selic (editor), UML Profile for Schedulability, Performance and Time, OMG Full Specification, formal/03-09-01 (2003).

[12] J. Skene, W. Emmerick, Model-driven performance analysis of Enterprise Information Systems, ENTCS 82(6), 2003.

[13] C.U. Smith, L.G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, 2002.

[14] ACM Proceedings of Workshop on Software and Performance, 1998-2005.