Resource Contention Analysis of Service-Based Systems through fUML-Driven Model Execution[*]

Martin Fleck[1], Luca Berardinelli[2], Philip Langer[1], Tanja Mayerhofer[1], and Vittorio Cortellessa[2]

[1] Business Informatics Group, Vienna University of Technology, Austria
{fleck, langer, mayerhofer}@big.tuwien.ac.at
[2] Dept. of Information Engineering and Computer Science and Mathematics, L'Aquila, Italy
{berardinelli, cortellessa}@univaq.it

Abstract. Model-driven software engineering not only enables the efficient development of software but also facilitates the analysis of non-functional properties (NFPs). As UML, the most adopted modeling language for designing software, lacks in formal execution semantics, current approaches translate UML models into dedicated analysis models, before NFPs can be computed. However, such transformations introduce additional complexity for the users and developers of analysis tools. To avoid this additional complexity, we show how the analysis of certain NFPs can be performed solely based on UML models without translating them into, e.g., queuing networks. Therefore, we leverage the execution semantics of fUML, a recent OMG standard, to gain execution traces from UML models and, based on these traces, compute indices, such as response time, taking into account the contention of resources, as well as different resource management configurations, such as balancing and scaling strategies.

1 Introduction

With the advent of model-driven software engineering, developers are empowered to raise the level of abstraction during the development using high-level models and to automatically generate executable code. This shift from code to models facilitates also the analysis of non-functional properties (NFPs) from early stages of its development [1].

UML [2] is currently the most adopted design modeling language whose extensibility, through UML profiles, lead to the emergence of several UML-based approaches for analyzing NFPs of the modeled software. However, due to the lack of a formal execution semantics of UML and the lack of UML-based tools for NFPs analysis, current approaches translate UML models into different kinds of analysis models, such as queuing networks (QN), for sake of performance analysis. Thus, a semantic gap between UML models and analysis models has to be bridged using often complex chains of model transformations before NFPs can be analyzed.

Although researchers have accomplished significant advances in transforming UML models in combination with applied UML profiles, such as MARTE [3], to dedicated

[*]This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859, and by the PRESTO ARTEMIS project (GA n. 269362).

analysis models, translational approaches suffer from some inherent drawbacks. Transformations have to generate analysis models that correctly reflect the heretofore informal semantics of UML models using concepts of the target analysis modeling language. Implementing such transformations is a complex task that requires deep knowledge not only of the semantics of UML and of the target analysis languages, but also of model transformation techniques, which hampers significantly the development and emergence of novel analysis tools. Even though transformations already exist, such transformation chains introduce inevitably an additional level of indirection, additional notations, and hence additional complexity, such as the consistent propagation of UML model changes to the analysis model and analysis results back on the UML model. This is a very relevant obstacle to the usability of analysis tools, because usually software developers are not trained in understanding formal languages applied for the analysis [4].

To address these drawbacks, France et al. [5] suggested to integrate analysis algorithms directly with the source modeling language, such as UML. Following this suggestion, we proposed in previous work [6] an approach for analyzing NFPs based on executing UML models directly. Instead of translating UML models into different notations, we showed how the execution semantics of the Foundational UML (fUML) [7] (i.e. a formally defined subset of UML standardized by the OMG) can be exploited to obtain model-based execution traces [8] from UML models and how these traces can be processed to compute NFPs, such as resource usage and execution time. As our previous work supported only the analysis of single independent execution traces, we could not consider the *contention of resources*. This aspect, however, is of uttermost importance when it comes to analyzing, for instance, the scalability of cloud-based applications on the IaaS layer or the thread contention in multicore systems.

In this paper, we address this limitation and extend our previous work to study the influence of resource contention on NFPs, such as *response time, throughput, and utilization*, that require the consideration of multiple overlapping executions. We enable this analysis within our fUML-based framework by obtaining *execution traces* from executing UML models that are annotated with the MARTE profile [3], attach timing information to these execution traces according to a *workload* specification, compute the *temporal relation* of these execution traces, and calculate performance indices that can so far be only obtained through translating UML models and performing the analysis based on other notations and formalisms, such as QNs.

As no transformation and no notation other than UML is involved, the presented framework is easily extensible with respect to the integration of additional analysis aspects. Thus, we further incorporated the analysis of *load balancing and scaling strategies* into our framework. Thereby, developers are equipped with methods for reasoning about optimal resource management configurations of the modeled components.

This paper is organized as follows. Section 2 briefly describes how we leveraged fUML in previous work to enable NFP analysis based on profile applications and execution traces and discusses its current limitations. In Section 3, we present an approach based on fUML execution traces for overcoming these limitations to also enable the analysis of NFPs taking resource contention into account. Section 4 showcases the applicability and benefits of our framework in a case study, before we discuss related work in Section 5 and conclude the paper in Section 6 with an outlook on future work.

2 Model Analysis with fUML

The fUML standard formally defines the *execution semantics* of a subset of UML. This subset consists of the structural and behavioral kernel of UML and comprises the most important modeling concepts for defining the structure of a system using *UML classes* and its behavior using *UML activities*. The fUML standard is accompanied by a Java-based reference implementation of a *virtual machine*, which allows to compute outputs from executing fUML activities with specified input parameters. While this enables executing fUML-compliant models and validating their execution output, a thorough analysis of a performed model execution is not possible. This prevents the model-based analysis of functional and non-functional properties of the modeled system.

To address the limitation, we extended the reference implementation of the fUML virtual machine in previous work [8] in order to capture *execution traces* and to provide them as additional output of a performed model execution. An execution trace provides the information necessary for analyzing the runtime behavior of the executed model. It captures information about the call hierarchy among executed activities, the chronological execution order of activities and contained activity nodes, the input provided to and the output produced by the activities and activity nodes, as well the token flow.

We may now exploit this rich information contained by the execution traces for performing further analyses of the system, such as performance analysis. However, we therefore require additional information, such as execution times of single activity nodes or activities. This information is usually provided by making use of dedicated UML profiles, such as MARTE [3]. Unfortunately, fUML does not support profiles. To address this issue, we presented a framework [6], which takes common UML models and applied profiles as input, seamlessly adapts those models to fUML for executing them, and transparently maps the resulting execution traces back to the level of UML, where the information on profile applications is again accessible. Thus, our framework enables the development of analysis components that leverage the well-defined semantics of fUML for capturing the runtime behavior of UML models in combination with the additional information from UML profiles. Based on this framework, we showed how performance analysis methodologies that are based on execution graphs [9] can be conducted directly on UML models and execution traces to aggregate demands of computing, storage, and communication resources.

3 Analyzing Resource Contention based on Execution Traces

To carry out a performance analysis that considers the *contention of resources*, we have to deal not only with single independent executions but multiple overlapping executions. This, however, is not possible yet as neither plain fUML nor our existing analysis framework allows to run competing executions of models concurrently within a shared runtime environment. Only sequential executions are supported so far, resulting in execution traces that are independent of each other. However, when analyzing software systems, performance indices concerning the contention of resources, such as *response time, utilization, and throughput*, are of utmost importance.

In this section, we show how we addressed this limitation to enable the analysis of resource contention based solely on UML models, profile application, and model

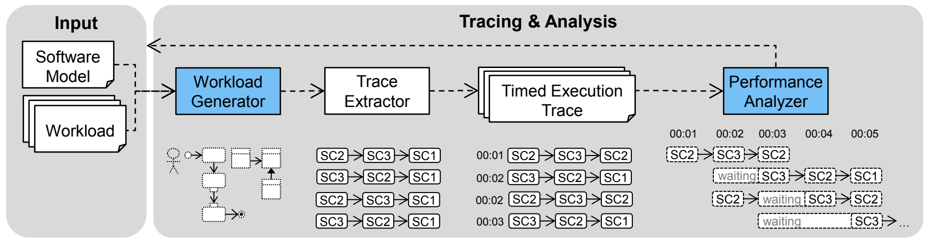

Fig. 1. Model-based performance analysis framework.

execution, without the need to translate the involved models into different notations and formalisms, as it is done in existing approaches. Note, however, that in this paper we consider software components as shared resources, whereas we plan to extend this work to platform resources in the future.

Overview. As proposed by Di Marco [10], we adopt the idea of considering software components as *service centers* [11]. A *job* has an arrival time specifying the point in time at which it enters the system and requires the *services* of different service centers. As long as a request is processed by a service center, further requests to this service center are stored in a queue until the service center is available. The next request from the queue is chosen, in our case, following the first-come, first-serve (FCFS) principle. Based on these concepts, mature algorithms are available to compute performance indices under resource contention.

For applying these concepts and algorithms to UML models directly, without translating them into dedicated performance models, we propose to: *(i)* trigger executions of UML models according to specific *workloads* for obtaining execution traces that reflect the runtime behavior of jobs (i.e., which services are requested in which order), *(ii)* annotate the arrival time to each of the resulting execution traces, and *(iii)* compute, based on known service times of consumed services, the temporal relation of concurrently running jobs (cf. Figure 1). Based on the temporal relation of executed jobs, we can step-wise derive their temporal overlaps and compute waiting times in each queue and, in further consequence, the overall response time, throughput, and utilization indices.

In addition, we introduce dedicated types of service centers that support *balancing and scaling strategies* to allow users of our framework to reason about optimal resource management configurations. In particular, a single service center may distribute incoming jobs to multiple instances of this service center according to certain strategies, as well as dynamically allocate and deallocate instances (*horizontal scaling*).

Input. The proposed framework takes as input a UML model stereotyped with MARTE. A summary of the stereotypes used for the software model (SW), hardware platform (HW), and workload model (WL) can be found in Table 1. The UML model contains the specification of the software structure and behavior, whereas MARTE is used to specify the system workload(s) as well as the performance characteristics of its structural and behavioral modeling elements.

The *software specification* consists of one or more class diagrams defining the structure and activity diagrams representing its behavior. Classes that should act as software

Table 1. MARTE stereotypes used in the framework.

View	Stereotype Name	Applied To	Used Tagged Values
HW	HwProcessor	Class	mips
SW	RtScalableUnit	Class	srPoolSize, queueSchedPolicy, scalingStrategy, timeToScale, scalingRange, scaleInThreshold, scaleOutThreshold, balancingStrategy
SW	GaStep	Activity	execTime
WL	GaScenario	Activity Diagram	cause, root
WL	GaWorkloadEvent	Activity	effect, pattern

service centers during the model execution (*RtScalableUnit*) have to be extended with information regarding: *(i)* the initial number of instances (*srPoolSize*), *(ii)* the scheduling policy for the incoming operation calls (*queueSchedPolicy*), *(iii)* the balancing strategy for selecting the instance that receives the next request (*balancingStrategy*), and, optionally, (iv) the rules for horizontal scaling (*scalingStrategy, timeToScale, scalingRange, scaleInThreshold, scaleOutThreshold*). Currently, no stereotypes in MARTE can represent both balancing strategies and scaling rules, thus we extended the existing stereotype *RtUnit* in this initial version of the framework to provide a set of pre-defined rules from which the modeler can choose, such as round robin or random balancing, and scaling based on the queue length.

In addition, the UML activities representing the software behavior (*GaStep*) have to be annotated with their respective execution times (*execTime*). These values may be either computed by estimating the complexity of behavioral units (e.g., number of executed instructions) in combination to the underlying platform characteristics (e.g., millions of instructions per second of *HwProcessor*), as done in our previous work [6,12], or obtained from existing benchmark services.

Alongside the structure and behavior of the software, the modeler has to specify the *workloads* in terms of UML activities that represent the expected interactions with the software. Such interactions start with a workload event (*GaWorkloadEvent*, e.g., a user interaction with the system) and a behavior scenario (*effect*) that is triggered by that event (*cause*). A behavior scenario (*GaScenario*) is a sequence of execution steps (i.e., calls of activities in the software models) that require the operations associated to service centers (that is, the *RtScalableUnit* classes). To specify how often a scenario is triggered, the modeler provides an arrival *pattern* for different types of workloads, such as periodic, open, or closed workloads.

Workload Generator. Once the UML model is provided, the analysis can be started. In a first step, a *workload generator* component reads the scenarios defining the software workload and automatically runs each of them once by executing the associated activities on top of the fUML virtual machine. From these executions, we obtain one execution trace for each scenario that captures the order in which services are requested as well as the execution time for each of the requests. In a next step, the traces are annotated with their arrival times as obtained from the inter-arrival times randomly generated from a probability distribution (e.g., exponential) according to the specified arrival pattern. This step results in a set of *timed execution traces*.

Performance Analyzer. The Performance Analyzer takes the timed execution traces with their execution time as input and performs operational analysis [11]. Special consideration is given to service centers having multiple instances, because they require balancing strategies to determine which instance will get the next request from the queue. Currently, only simple balancing and scaling strategies, such as round robin, random balancing, and scaling based on queue length, are implemented in our prototype. However, additional strategies can be easily added. Once the analysis is completed, the resulting performance values are annotated in the UML model using a *GaAnalysisContext* stereotype on the respective UML elements. This allows the user to view the results directly in the editor used for UML modeling. The obtained performance values include, among others, the waiting time and service time for the scenarios and utilization and throughput for the service centers, i.e., instances of the corresponding UML classes. Furthermore, it is possible to calculate the performance values for a specific point in time or for a given time frame within the complete simulation time. Thus, detailed graphs can be generated that show the evolution of the performance values to provide a convenient overview. The validity of our performance analyzer has been successfully checked by comparing its result values with the result values of an established QN analysis tool [13]. Detailed results can be found on our project website [12].

4 The PetStore Case Study

To evaluate the feasibility and applicability of our approach, we realized a prototypical implementation [12] and performed a case study. This case study concerns a *PetStore*, which is a simple online store where customers can register, log in, browse a catalog of pets, add them to their shopping cart, and place orders. The components of the *PetStore* are as follows. The *Application Controller* is responsible for handling the user interaction and can interact with three other software services, namely *CustomerService*, *CatalogService*, and *OrderService*, that manage the customers, items, and orders, respectively. These services, in turn, have access to an *EntityManager* that provides operations to persist, retrieve, and delete *PetStore* data.

Input. As underlying hardware platform for the *PetStore*, we assume that each component runs on the same execution host having its own computing resource, i.e., CPU with the capability to handle a certain amount of million instructions per second (*mips*). Hence, as mentioned before, the resource contention is limited here to software components. The *PetStore* itself is modeled using UML class diagrams for the structure and activity diagrams for the behavior. Operations are annotated with their execution times (*GaStep*) which were determined by creating an *overhead matrix* that combines an estimated number of high-level instructions executed for each operation call as well as the capability of the CPU, as described by Smith and Williams [9]. In this case study, we want to analyze a typical online shopping workload consisting of the *BuyItem Scenario*, which represents a user that logs into the *PetStore*, searches for a specific item, adds this item to the shopping cart and confirms the order. We assume this to occur on

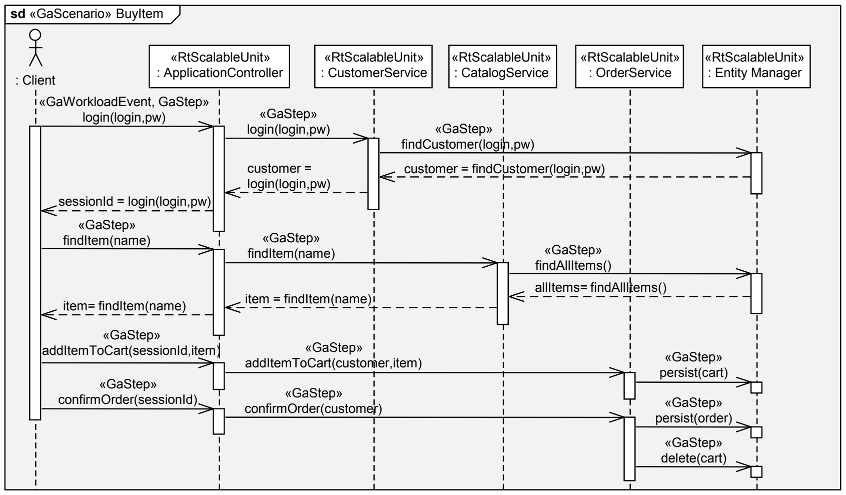
sd «GaScenario» BuyItem
: Client
«RtScalableUnit» : ApplicationController
«RtScalableUnit» : CustomerService
«RtScalableUnit» : CatalogService
«RtScalableUnit» : OrderService
«RtScalableUnit» : Entity Manager
«GaWorkloadEvent, GaStep» login(login,pw)
«GaStep» login(login,pw)
«GaStep» findCustomer(login,pw)
customer = findCustomer(login,pw)
customer = login(login,pw)
sessionId = login(login,pw)
«GaStep» findItem(name)
«GaStep» findItem(name)
«GaStep» findAllItems()
allItems= findAllItems()
item = findItem(name)
item= findItem(name)
«GaStep» addItemToCart(sessionId,item)
«GaStep» addItemToCart(customer,item)
«GaStep» persist(cart)
«GaStep» confirmOrder(sessionId)
«GaStep» confirmOrder(customer)
«GaStep» persist(order)
«GaStep» delete(cart)

Fig. 2. The *PetStore BuyItem Scenario*

average every two seconds, exponentially distributed, which is annotated as pattern in the triggering event of the scenario. Figure 2 depicts the *BuyItem Scenario* including the involved classes and called operations. Each lifeline corresponds to a service center instance while each asynchronous message corresponds to the invocation of an operation on the receiving lifeline. Note that in our model the service center annotations are applied to a class level and not to an instance level, so all instances of the same class share the same performance characteristics. The scenario as well as its setup, e.g., the available pets or existing user accounts, are modeled using fUML activity diagrams.

Due to space restrictions, we are not able to show all models in detail. We therefore kindly refer the reader to the project website [12], where all models can be downloaded.

Analysis. Using the proposed framework introduced in Section 3, it is possible to compute the utilization, throughput, and response time of all jobs for the overall workload, as well as the minimum, average, and maximum waiting time and service time of the jobs for each scenario. Besides, we calculate the idle time, busy time, utilization, throughput, as well as the minimum, average, and maximum queue length for each service center. The computed results are annotated in the UML model using the *contextParam* tagged value of the *GaAnalysisContext* stereotype. Additionally, we generate graphs showing the evolution of those indices over time. Using the defined UML model as input, we can reason about different configurations on software service level and explore the effect of different balancing and scaling strategies. For simplicity, we consider four configurations in our example and focus on the utilization of service centers within the system, as well as the overall execution time.

For all configurations we apply the same workload within the same time frame of 6 seconds. The result for each single configuration is depicted in Figure 3. As baseline, the first configuration (*Base*) only considers one instance per service center and uses neither balancing nor scaling, resulting in a average execution time of about 87 seconds. From these results we can identify the *EntityManager* as the bottleneck of the application: the *EntityManager* has a very high utilization and blocks an optimal utilization of the

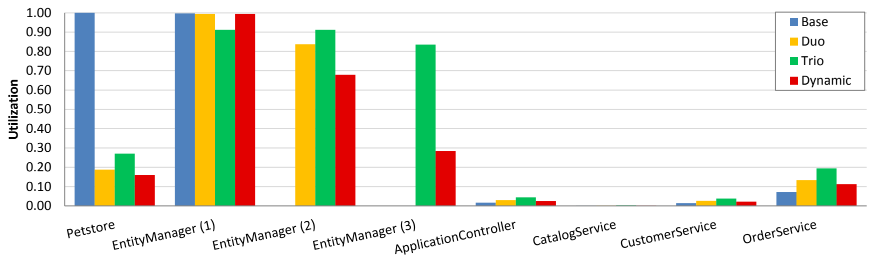

Fig. 3. The Utilization of the *PetStore BuyItem Scenario*

other components. This is not surprising considering that the *EntityManager* is needed for almost every operation.

Trying to improve this result, we introduce one (*Duo*) and two (*Trio*) additional *EntityManager* instances and balance the requests between these instances using a round robin strategy. Figure 3 shows the utilization of these additional instances, identified by the number in the parenthesis. We can see that, in our example, multiple instances of the *EntityManager* center can reduce the execution time to almost a half or third and, hence, increase the utilization of the other service centers. However, more instances usually also imply more costs, thus making the number of instances a tradeoff between cost and performance.

In the fourth configuration (*Dynamic*), we vary the number of *EntityManager* instances dynamically instead of choosing a fixed number. We introduce the following horizontal scaling strategy: whenever the average queue length of the *EntityManager* is larger than 1.2, a new *EntityManager* instance should be allocated, and whenever the average queue length is lower than 0.6, an instance should be removed. For the sake of experimentation, the time needed for adding and removing instances is set to 100ms and we allow the number of instances to range from one to three, starting with one instance. The results for this configuration show that two additional instances are created during the run time, but neither of them can reach a high utilization, indicating that they might have been allocated too late. In comparison with the previous two configurations, we can further see that horizontal scaling yields no real benefit in our example. Possible reasons for this could be that the specified workload has little time between two jobs and that the average queue length does not reflect the changes made through scaling fast enough, resulting in a quick allocation and deallocation of many instances in a short period of time.

However, despite these results, this case study illustrates that NFPs concerning the resource contention can be analyzed solely based on UML models and execution traces with our framework. Moreover, it shows that our framework is extensible as it allows to consider further concerns, such as the analysis of optimal configuration of balancing and scaling strategies, which can be easily integrated with our approach.

5 Related Work

Due to the previous lack of semantics in UML, many UML model-based analysis approaches (cf. Balsamo et al. [4] for a survey) have implemented dedicated model transformations to specific performance models that can be used as input for existing analysis tools such as JMT [13]. This eventually led to the introduction of common performance model interchange formats, such as PMIF [14] or CSM [15], to reduce the effort for transforming UML models to performance models and for integrating new methodologies with existing tools. However, due to the fact that many analysis tools existed before the introduction of the interchange formats, there is still limited support for these intermediate formats in analysis tools [16]. Other methodologies such as Palladio [17] overcome this problem by introducing their own proprietary modeling notation and their own integrated analysis tools. However, all mentioned approaches require a translation of UML models into corresponding modeling notations, before the analysis can be performed.

In the future, we intend to extend our work to the analysis of cloud-based systems. However, no consensus on the right set of models, languages, model transformations, and software processes to combine MDE and cloud computing has yet been reached [18]. Nevertheless, there are already notable cloud analysis tools available, such as CloudSim [19] and its extensions, NetworkCloudSim [20] and DynamicCloudSim, but they either adopt programming languages rather than software modeling notations to specify simulations or use existing generic simulation platforms like OMNET++ as done in iCanCloud [21].

6 Conclusion and Future Work

In this paper, we have proposed an approach for enabling UML-based performance analysis, taking into account the contention of software resources, without needing to transform UML models into performance models. We leverage the well-defined semantics of fUML for obtaining execution traces that represent the requested software services according to a user-specified workload. We propose to annotate execution traces with timing information and compute their temporal overlaps, as well as known performance indices, such as response time and throughput. Beside the extension of our framework to resource contention, we have considered peculiarities of dynamic resources provisioning, as we integrate capabilities of balancing and scaling strategies.

In the future, we aim to build a framework that enables users to analyze different aspects of cloud-based software. Thus, we plan to extend the concept of shared resources from software components to platform components and integrate more complex balancing and scaling strategies to improve performance analysis. Furthermore, we intend to support the analysis of costs based on the calculated timing information and dedicated pricing models. Additionally, we plan to analyze the scalability of our approach with different case studies and to compare the effectiveness of our approach with traditional approaches from a users perspective, e.g., usability, and a developers perspective, e.g., the complexity of integrating further analysis methods, with controlled experiments.

References

1. D. C. Petriu, M. Alhaj, and R. Tawhid. Software Performance Modeling. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *LNCS*, pages 219–262. Springer, 2012.

2. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011. Available at: http://www.omg.org/spec/UML/2.4.1.
3. Object Management Group. UML Profile for MARTE, Version 1.1, June 2011. Available at: http://www.omg.org/spec/MARTE/1.1.
4. S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE TSE*, 30(5):295–310, 2004.
5. R. B. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proc. of the Workshop on the Future of Software Engineering (FOSE) @ ICSE*, pages 37–54, 2007.
6. L. Berardinelli, P. Langer, and T. Mayerhofer. Combining fUML and Profiles for Non-Functional Analysis Based on Model Execution Traces. In *Proc. of the 9th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA)*. ACM, 2013.
7. Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0, February 2011. Available at: http://www.omg.org/spec/FUML/1.0.
8. T. Mayerhofer, P. Langer, and G. Kappel. A Runtime Model for fUML. In *Proc. of the 7th Workshop on Models@run.time (MRT) @ MoDELS*, pages 53–58. ACM, 2012.
9. C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, volume 1. Addison-Wesley, 2002.
10. A. Di Marco. *Model-based Performance Analysis of Software Architectures*. PhD thesis, University of L'Aquila, 2005. http://www.di.univaq.it/adimarco/thesis/thesis-final-web.zip.
11. E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, 1984.
12. Business Informatics Group. Model Execution Website. http://modelexecution.org.
13. M. Bertoli, G. Casale, and G. Serazzi. JMT - Performance Engineering Tools for System Modeling. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.
14. C. U. Smith, C. M. Lladó, and R. Puigjaner. Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability. *Performance Evaluation*, 67(7):548–568, 2010.
15. D. B. Petriu and M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and Systems Modeling*, 6(2):163–184, 2007.
16. C. U. Smith, C. M. Lladó, and R. Puigjaner. PMIF Extensions: Increasing the Scope of Supported Models. In *Proc. of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 255–256. ACM, 2010.
17. S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.
18. H. Brunelière, J. Cabot, and F. Jouault. Combining Model-Driven Engineering and Cloud Computing. In *Proc. of 4th Workshop on Modeling, Design, and Analysis for the Service Cloud (MDA4ServiceCloud) @ ECMFA*, 2010.
19. R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.
20. S. K. Garg and R R. Buyya. NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations. In *Proc. of the 4th IEEE International Conference on Utility and Cloud Computing (UCC)*, pages 105–113, 2011.
21. A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente. iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.