fUML-Driven Performance Analysis through the MOSES Model Library*

Luca Berardinelli, Vittorio Cortellessa

Dept. of Information Engineering, Computer Science and Mathematics,
University of L'Aquila, 67100 L'Aquila, Italy
{luca.berardinelli,vittorio.cortellessa}@univaq.it

Abstract. The growing request for high-quality applications for embedded systems demands model-driven approaches that facilitate their design as well as the verification and validation activities.
In this paper we present MOSES, a model-driven performance analysis methodology based on Foundational UML (fUML). Implemented as an executable model library, MOSES provides data structures, as Classes, and algorithms, as Activities, which can be imported to instrument fUML models and then to carry out the performance analysis of the modeled system through fUML model simulation. An industrial case study is provided to show MOSES at work, its achievements and its future challenges.

Keywords: fUML, Model-Driven Performance Analysis, Tool Support

1 Introduction

Evolution in the industrial process development of real time and embedded systems (RTES) has to face new challenges, especially in the design process.

By nature, RTES are constrained by the limited amount of resources available (e.g., time, power, size) and these constraints need to be considered throughout the engineering process. Allocation of application functions on execution platforms, and the related consequences on resource usages need to be carefully addressed during the design stages.

However, the software for RTES is traditionally developed adopting a code-and-fix approach, thus neglecting design as well as verification and validation (V&V) activities. This approach may result in software components that miss functional and/or extra-functional requirements (e.g., performance), so compromising the system deployment on the hardware platform that is usually co-designed with the software [1].

Model-Driven Engineering (MDE) and Component-Based Software Engineering (CBSE) paradigms may play a capital role in the RTES domain by emphasizing, on one side, the adoption of models as the main design artifacts throughout the whole development process, and on the other side the design and implementation of complex systems through reusable software components.

* This work is partially supported by the EU-funded VISION ERC project (ERC-240555), and by PRESTO ARTEMIS project (GA n. 269362).

RTES is the application domain of MOSES (MOdeling Software and platform architEcture in UML for Simulation-based performance analysis), a model-driven methodology based on executable UML models.

MOSES [2] was originally devised for UML 1.x Real Time models and later ported to UML 2.x. In this paper, MOSES is re-designed from scratch to make it compliant with the Foundational UML standard (fUML) [3], a strict UML subset by the Object Management Group, provided with its own executable semantics and virtual machine.

This work has been conducted in the context of the PRESTO project (imProvements of industrial Real-time Embedded SysTem development prOcess) [4], which aims at improving the RTES development process with model-driven techniques while considering industrial constraints like, for example, a smooth integration in current development processes. In this regard, it may happen that existing, *trusted* Commercial-Off-The-Shelf components (COTS) lack of model-based specifications (e.g., UML models), thus hindering their reuse within MDE approaches. In PRESTO such limitations have been tackled by exploiting execution traces generated by test execution during the software integration phase. Such traces may help developers to narrow the boundary of the system that undergoes V&V activities by limiting the analyses to tracked components only. In this context, MOSES has been re-designed to represent traces in fUML, and to exploit them for modeling the system workload, with the aim of performance analysis based on fUML model simulation.

In this paper we present the new design and performance analysis capability of MOSES with the help of an industrial case study.

The rest of the paper is organized as follows. Section 2 provides a quick background on fUML. Section 3 details the proposed case study and its fUML model. Section 4 introduces MOSES and its modeling and performance analysis capabilities based on the fUML semantics. Section 5 shows MOSES in action on the case study. Finally, Section 6 discusses current limitations and future challenges for MOSES also with respect to related work. Section 7 concludes the paper.

2 The Foundational UML

Both MOSES and the case study have been modeled in fUML[3]. It defines the operational semantics of a strict, computationally complete UML subset that includes *Classes*, *Common Behaviors*, *Activities*, and *Actions* language units. In essence, fUML enables the execution of UML models where structural elements are classes with their own properties, operations, and associations while the behavioral specifications (e.g., operation body) are modeled through UML activities.

The fUML standard goes along with a Java-based reference implementation[1] of an *fUML virtual machine* (fUML VM). Free open source and commercial UML modeling tools exist that embed this reference implementation within their

[1] `http://fuml.modeldriven.org`

modeling environments, like Papyrus and MagicDraw[2]. We have adopted the latter as the main modeling environment and its Cameo Simulation Toolkit plug-in to enable the model simulation.

Since fUML does not introduce any heavyweight extension, any fUML model is UML-compliant. At run time, the fUML VM generates a so-called *instance model* and ignores the non-executable part (including annotations from UML profiles [5]). InstanceSpecifications, Links, and Slots elements are generated within the instance model as the run-time counterparts of Classes, Associations and Properties, respectively. In this respect, the execution of fUML activities adds, deletes or modifies elements of the instance model.

3 Case Study: Indoor Positioning System

The case study that we consider in this paper concerns the SW/HW development of an Indoor Positioning System (IPS) based on a Mobile Ad Hoc NETwork (MANET). MANETs are self-configuring and self-healing networks, which do not require any pre-existing infrastructure or centralized control. Their nodes are mobile, connected by wireless links, then the network topology is very dynamic.

With regard to the software part, IPS reuses the Optimized Link State Routing (OLSR) [6] as its IP routing protocol. OLSR is optimized for MANETs because it minimizes the broadcast of control messages by forcing their flow through selected nodes, a.k.a. multipoint relays (MPRs). OLSR is a COTS: both a Request for Comments (RFC3626)[6] by the Internet Engineering Task Force, and standard implementations (OLSR Daemon, *olsrd* [3]) exist for such protocols. The RFC3626 modularizes OLSR into *core functionalities* (Neighbour Sensing, Multipoint Relaying, Link-State Flooding) and defines three types of *control messages* (HELLO, Topology Control TC, and Multiple Interface Declaration MID) whose contents are stored in eight different *information repositories*. The hardware platform consists of IPS nodes including two main components: an ATMEL ATZB-900-B0 module [4] that sustains, with its transceiver, the signaling among nodes, and ii) an OMAP L138 module[5], embedding a general purpose ARM CPU and a Digital Signal Processor (DSP), that sustains physical, medium access control, and network layers of the communication network. The OLSR daemon is deployed on the OMAP.

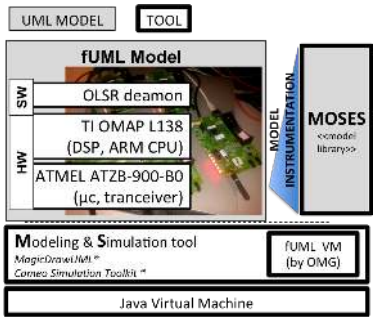

Fig. 1: MOSES and its surrounding environment.

Figure 2b shows an excerpt of both the software architecture and the hardware platform of the IPS fUML Model. The abstraction level (i.e., what is ex-

[2] www.papyrusuml.org/,www.nomagic.com/products/magicdraw.html
[3] IPS software: http://www.olsr.org/.
[4] http://www.atmel.com/Images/doc8227.pdf
[5] http://www.ti.com/product/omap-l138

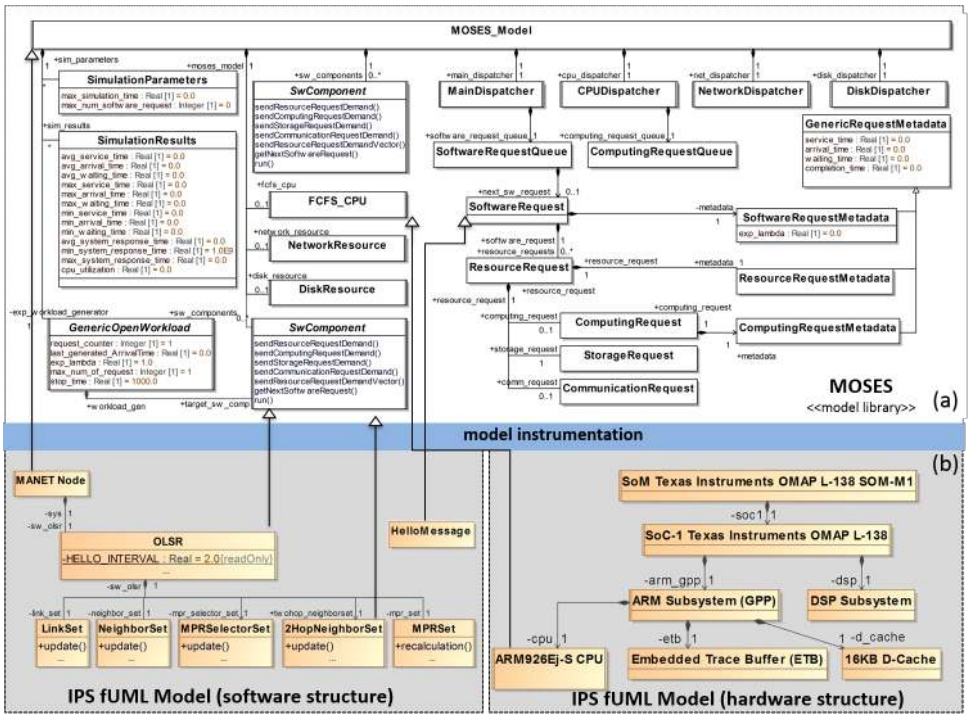

Fig. 2: Excerpts of the MOSES library architecture (a), the instrumented MANET software and hardware structures (b).

plicitly modeled and what is left out as external environment) is chosen in accordance with the execution traces obtained by the case study provider, namely Thales Italy, then runs the OLSR daemon on both virtualized environments and prototypical test-beds of several IPS nodes [4]. In particular, the tracked OLSR execution traces involved i) five (over eight) `OLSR` information repositories, namely `LinkSet`, `NeighborSet`, `MPRSelectorSet`, `2HopNeighborSet`, and `MPRSet`, and ii) the link sensing functional capability (over three), all described in the RFC3626 [6]. Moreover, since the link sensing is carried out through the flooding of `HELLO Message`s, the other kinds of messages (TC, MID) are not part of the IPS fUML Model.

The hardware part may include all the computing, storage and communication resources embedded on the `ATMEL ATZB-900-B0` and `OMAP L138` modules. The latter is modeled through compound classes in Figure 2b. In particular, we concentrate only on computing resources, i.e., the `ARM926Ej-S CPU`, which processes the software requests arriving at the `OLSR` component deployed on each `MANET Node`.

The IPS software behavior is represented through hierarchical fUML activities. Top level activities are meant to reproduce, at simulation time, the software execution traces (traceable as Message Sequence Charts [7]) through call operation actions (Figure 3a) that, in turn, are further detailed with activities assigned to operation methods, as shown for the `LinkSet::update()` in Figure 3b.

4 The MOSES Model Library for Performance Analysis

MOSES[6], originally implemented within UML RealTime [2], has been here redesigned as an fUML model library [3] (see Figure 1) whose data structures (including analysis results) and algorithms are represented by UML Classes and Activities, respectively. As a consequence, the main benefit of MOSES has been preserved throughout its evolution process, that is allowing performance analysis while avoiding translational approaches to different external notations and related technological spaces [8]. This is achieved by integrating the performance analysis algorithms and results directly within the modeling language used in systems development, as suggested in [9].

MOSES is meant to instantiate an intermediate layer between the components of a software architecture and those of the underlying platform. Such layer is in charge to model and deliver additional data for the sake of performance analysis.

The MOSES architecture is shown in Figure 2a. A `MOSES Model` is composed by `SwComponent`s running on a platform that provides different kinds of hardware resources, like `FCFS CPU`s, `DiskResource`s, `NetworkResource`s. `SwComponent`s receive inputs from the surrounding environment in terms of one or more `Software Request`s that, in turn, include one or more `ResourceRequest`s (i.e. `Computing Request`s, `StorageRequest`s, and `CommunicationRequest`s) to specific hardware resources of the underlying platform (`FCFS CPU`s, `DiskResource`s, `Network Resource`s, respectively).

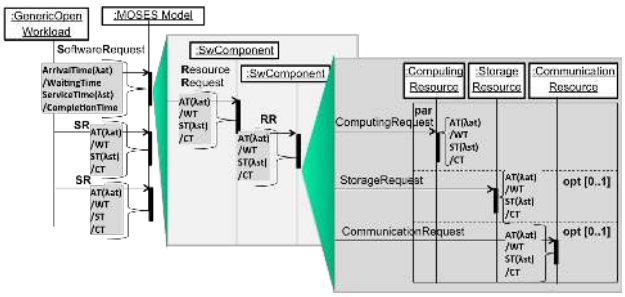

Fig. 3: MOSES and its surrounding environment.

The fUML VM *as is* does not provide any data structure or algorithm specific for performance analysis purposes. In particular, as remarked in [10], the fUML VM does not support the notion of time, which is fundamental for any kind of performance analysis. To cope with this limitation, MOSES characterizes each request with a set of metadata (see Figure 3) that are Arrival Time (AT), Service Time (ST), Waiting Time (WT), and Completion Time (CT), filled during the simulation. In particular, MOSES represents ATs and STs indices as exponentially distributed random variables, with distinct *lambda* parameters. The simulation engine uses such variables to sample i) the average amount of time between the arrival of consecutive requests and ii) the average amount of resource requested. In addition, the exponential distribution guarantees the memoryless property between the corresponding events, i.e. the arrivals of *SoftwareRequest*s and resource usages, respectively.

[6] Further details at `http://sealabtools.di.univaq.it/tools.php`

The other two indices, WT and CT, result from the simulation process and thus they are modeled as derived properties. In MOSES, the combination of these four indices (ATs, WTs, STs, and CTs) allows to simulate resource sharing among software components and, then, to calculate performance indices like system response time and resource utilizations under concurrency.

Figure 3 shows, with the help of an UML-like Sequence Diagram[7], how the aforementioned MOSES library elements interact once instantiated and simulated by the fUML VM. MOSES generates AT(λ_{at})s and ST(λ_{st})s and derives /WTs and /CTs, accordingly, for each execution occurrence over lifelines representing active objects at simulation time (from left to right, the whole `MOSES Model`, its constituting `SwComponent`s and platform resources of their execution hosts).

In MOSES, all kinds of requests are collected and managed by a hierarchical set of *dispatchers*. A top-level `MainDispatcher` is in charge of splitting each `SoftwareRequest` into one or more `ResourceRequest`s addressed to different components. Each `ResourceRequest` is further split in hardware-specific requests sent to hardware-level dispatchers, namely `CPUDispatcher`, `DiskDispatcher`, and `NetworkDispatcher`, which forward each specific request (`ComputingRequest`, `StorageRequest`, `CommunicationRequest`) to the proper hardware resource.

It is worth noting that, in MOSES, we assume that each `SoftwareRequest` always implies a computing request and, optionally, further storage and communication requests.

Finally, MOSES also provides i) workload generators of `SoftwareRequest`s (`GenericOpenWorkload`), and ii) data structures to manage requests, like dispatchers' queues (`SoftwareRequestQueue`, `ComputingRequestQueue`), and to store analysis parameters and results (`SimulationParameters`, `SimulationResults`).

5 Performance Analysis of MOSES Models

The performance analysis of MOSES models is carried out on top of MOSES intermediate layer which, in turn, exploits the simulation capabilities of the fUML VM. The correct instantiation of the MOSES intermediate layer at simulation time requires the *wiring* of the MOSES library with the user defined fUML Model through *model instrumentation* (see Figure 1). This modeling step is realized in two consecutive steps: i) the identification of software and hardware resources on the user-defined fUML Model by establishing their generalization to the corresponding MOSES classes, and ii) the behavior extension of the identified software/hardware components through MOSES-specific actions.

The former step is shown in Figure 2. The system boundary is represented by `MANET Node`, within the IPS network, which receives HELLOs from its neighbors. Therefore, a `MANET Node` corresponds to a `MOSES Model`, whose generic `SwComponent`s are concretely represented by the `OLSR` and its information repositories. The `ARM CPU` is the shared hardware resource that schedules the computing requests generated by the received HELLOs following a First-Come First-Served (FCFS) policy. The second step is exemplified in Figure 4 for the `LinkSet`

[7] This is not part of the MOSES library but used for explanatory purposes.

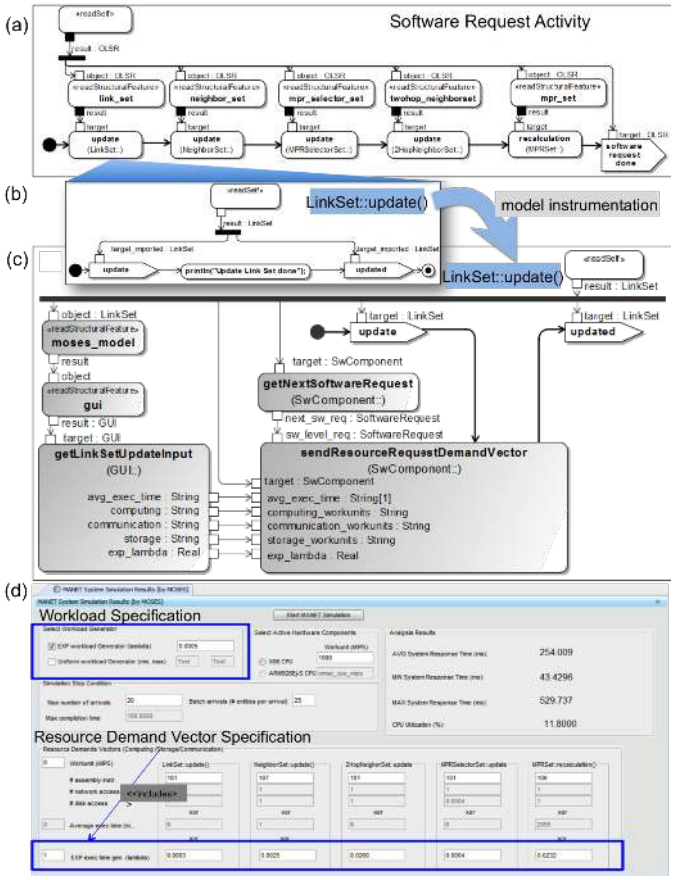

Fig. 4: OLSR software requests (a), LinkSet::update() before (b) and after model instrumentation by MOSES (c), and MOSES GUI (d).

`::update()` operation whose generic behavior (Figure 4b) is instrumented with MOSES-specific (gray) actions that are in charge of: i) scheduling software requests (`getNextSoftwareRequest`, ii) retrieving performance parameters from a case study-specific GUI (shown in Figure 4d), and iii) generating the software demands (`sendResourceRequestDemandVector`)) to shared platform resources. In our case study, for sake of illustration, we have limited the set of shared resources to the `ARM CPU` and, therefore, demands are quantified in CPU time (i.e., the service time ST, in Figure 3).

Given a suitably instrumented fUML model, the next step is to define the performance scenario, i.e. software request workload(s), performance requirement(s) and index(es).

For IPS we aim at estimating the utilization of the `ARM CPU` of a single `MANET Node` with 25 neighbors. The workload is represented by the receiving node processing of HELLO messages. Each message generates five consecutive operation calls to five distinct `OLSR`'s information repositories, as modeled through the executable activity in Figure 4a.

A successful scenario is determined by: i) an `ARM CPU` load lower then 50%, and ii) a response time, for each HELLO message, lower than 40 milliseconds . This latter threshold is determined by dividing the `OLSR` HELLO_INTERVAL [6] (set to 2 seconds) by the maximum system size that is set to 25 nodes. The violation of these performance requirements can cause the reshaping of the network or the choice of a more powerful CPU for the `MANET Node`.

Through three separated panels on the MOSES GUI (shown in Figure 4d), the MOSES user can provide inputs and observe the analysis results.

The Workload Specification (panel) characterizes the *arrival process* of HELLOs. The AT between two consecutive HELLOs is represented as an exponentially distributed random variable whose λ parameter is set to 0.0005 seconds.

Similarly, the Resource Demand Vector Specification (panel) groups the parameters that determine the ST of component operations in five sets, one for each operation invoked in the activity in Figure 3a. The STs can be *assumed*, in case of "what-if" analyses, or *measured* on software running on a predefined platform, if available. In our case study, we collected[8] the average execution times of the involved operations by executing the olsrd daemon on a real UWB node.

Table 1: Timing parameters for components' operations.

Component::Operation	$Avg(ST)_{1SR}$ (ms)	$Avg(ST)_{25SR}$ (ms)	$\lambda_{25} = (1/Avg(ST)_{25SR})$
LinkSet::update()	149.00	3725.00	$3.0E-4$
NeighborSet::update()	15.86	396.50	$2.5E-3$
MPRSelectorSet::update()	97.96	2449.00	$4.0E-4$
2HopNeighborSet::update()	1.54	38.50	$2.6E-2$
MPRSet::recalculation()	1.72	43.00	$2.32E-2$

Table 2: Performance Analysis Results.

Performance Index	**Required**	**Estimated**
CPU Utilization (%)	< 40	11.8
Max System Response Time (single HELLO - batch of 25 HELLOs) (ms)	80 - 2000	21.19 - 529.74

The last column on Table 1 lists the *lamba* parameters used to generate STs of component operations that, like ATs of HELLOs, are obtained from exponentially distributed random variables. Moreover, we model a *batch arrival process* of 25 HELLOs, that is, 25 software requests arriving at the same time and then sharing the same AT value. In accordance with this assumption, we suitably multiply the measured average STs ($Avg(ST)_{1SR}$) by the batch size 25 ($Avg(ST)_{25SR}$) and obtain the corresponding λ parameters (λ_{25}) to associate to demand vectors on the MOSES GUI. Finally, for this paper experiments we have stopped the simulation after the arrivals of 20 batches, for a total of 500 HELLOs.

MOSES analysis capabilities are currently limited to the calculation of two performance indices: i) the System Response Time (average, minimum, and maximum in milliseconds), which corresponds to the difference between the AT of a batch of HELLOs on the UWB node interface and the CT of whole batch, and ii) the CPU Utilization, i.e., the percentage of time (i.e. the CT of the latest

[8] through a software *profiler* available at `www.qnx.com`

batch) that the CPU spends in processing batches (obtained from WTs and STs of each batch).

The performance requirements outlined at the beginning of this section and the obtained results are listed in Table 2. For sake of this paper experimentation, we consider the results satisfactory even though further investigation and more complex performance scenarios shall be simulated in future.

6 Related Work

MOSES leverages fUML to enable performance analysis within the fUML technological space, without the need of translations to external notations, as traditionally approached in the performance analysis domain [11].

This paper is part of a broader research effort towards extra-functional analyses based on fUML model simulation[5,12,13]. In [12], we proposed the performance analysis of mobile agents for wireless sensor network in Agilla, that is a domain specific programming language whose behavioral units (namely *patterns*) and data structures are modeled as a reusable fUML library. In [5] and [13] we devised an Eclipse-based translational approach that combines fUML with profiles for post-simulation performance analysis of fUML *model execution traces*.

These approaches still suffer from i) the scarce availability of reusable *core* executable model libraries (e.g., common data structures like queue or stack), thus demanding a huge modeling effort to fUML modelers and, even more, ii) fUML VM design deficiencies that still limit the adoption of fUML for V&V analyses [10,14,15]. The former limitation may take advantage of the Action Language for fUML (Alf), an OMG *native* scripting language for UML behaviors. Alf can be used to specify large and complex UML Activities (as those realized for the MOSES library). Regarding the latter limitation, [10] and [14] both propose to redesign the fUML *execution model* [9] to support testing and debugging of fUML models [10], and concurrency, synchronization, and scheduling capabilities [14]. In [15], the authors proposed an extension of the fUML semantics via a fUML library (i.e., without the need of modifying the current fUML VM implementation) that is meant to be reused for a more efficient design of simulation frameworks based fUML, like MOSES. Finally, scalability issues may originate from the inherent nature of fUML libraries. Indeed, MOSES can be seen as a layered tool where all its functionalities run within a hosting UML modeling environment, which, in turn, run atop a Java Virtual Machine (see Figure 1). This layered infrastructure may cause scalability issues for analysis tools, like MOSES, running on the topmost layer. In this respect, we noticed that the simulation speed decreases while augmenting the number of arrivals. However, we used the fUML VM as a black box component as embedded in Cameo Simulation Toolkit. Alternative fUML VMs [10] should be tested to validate MOSES against particular VM's implementation biases. Assessing the maturity level of fUML and of available VMs is out of scope of this paper and left as future work.

[9] That is a UML model from which the fUML VM is generated.

[10] See `http://modeling-languages.com/list-of-executable-uml-tools/`

7 Conclusion

In this paper we presented MOSES, a methodology and tool for performance analysis realized as an executable fUML library.

MOSES has been redesigned within the JU Artemis PRESTO project. The new MOSES aims at reproducing the behavior of software execution traces at simulation time, in order to validate their performance requirements. An industrial case study has been provided to show MOSES at work and outline achievements and future challenges. As future work, we plan to further investigate the limitations of current and future fUML VM implementations (such as scalability issues) to extend the performance analysis capabilities of the MOSES library as well as applying similar fUML-driven approaches to the analysis of different extra-functional properties (e.g., reliability).

References

1. J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proc. of the IEEE*, 100(Special Centennial Issue):1411–1430, May 2012.
2. V. Cortellessa, P. Pierini, R. Spalazzese, and A. Vianale. Moses: Modeling software and platform architecture in uml 2 for simulation-based performance analysis. In *QOSA*, volume 5281 of *LNCS*, pages 86–102. Springer, 2008.
3. OMG. Semantics of a Foundational Subset for Executable UML Models, 2011.
4. PRESTO Consortium. imProvements of industrial Real-time Embedded SysTem development prOcess, http://www.presto-embedded.eu/, June 2014.
5. L. Berardinelli, P. Langer, and T. Mayerhofer. Combining fUML and profiles for non-functional analysis based on model execution traces. In *QoSA*, 2013.
6. IETF. OLSR 3626, http://tools.ietf.org/html/rfc3626.
7. E. Gaudin and E. Brunel. Property verification with MSC. In *SDL Forum*, pages 19–35, 2013.
8. J. Bézivin. Model driven engineering: An emerging technical space. *Generative and transformational techniques in software engineering*, pages 36–64, 2006.
9. R.B. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering*, pages 37–54, 2007.
10. Y. Laurent, R. Bendraou, and M.P. Gervais. Executing and debugging UML models: an fUML extension. In *Proc. of ACM Symposium on Applied Computing*, SAC '13, pages 1095–1102, New York, NY, USA, 2013. ACM.
11. V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis.* Springer, 2011.
12. L. Berardinelli, A. Di Marco, and S. Pace. fUML-Driven Design and Performance Analysis of Software Agents for Wireless Sensor Network. In *Software Architecture*, volume 8627 of *LNCS*, pages 324–339. Springer, 2014.
13. M. Fleck, L. Berardinelli, P. Langer, T. Mayerhofer, and V. Cortellessa. Resource contention analysis of service-based systems through fUML-driven model execution. *Proc. of NiM-ALP*, page 6, 2013.
14. A. Benyahia, A. Cuccuru, S. Taha, F. Terrier, F. Boulanger, and S. Gérard. Extending the standard execution model of UML for real-time systems. In *IFIP Conf. on Distributed and Parallel Emb. Sys. (DIPES'10)*, pages 43–54. Springer, 2010.
15. J. Tatibouet, A. Cuccuru, S. Gérard, and F. Terrier. Principles for the realization of an open simulation framework based on fuml (wip). In *DEVS Integrative M&S Symposium (DEV'13)*, pages 1–6. SCS, 2013.