Combining fUML and Profiles for Non-Functional Analysis Based on Model Execution Traces

Luca Berardinelli
Università dell'Aquila
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica
L'Aquila, Italy
luca.berardinelli@univaq.it

Philip Langer
Business Informatics Group
Vienna University of
Technology
Vienna, Austria
langer@big.tuwien.ac.at

Tanja Mayerhofer
Business Informatics Group
Vienna University of
Technology
Vienna, Austria
mayerhofer@big.tuwien.ac.at

ABSTRACT

For developing software systems it is crucial to consider non-functional properties already in an early development stage to guarantee that the system will satisfy its non-functional requirements. Following the model-based engineering paradigm facilitates an early analysis of non-functional properties of the system being developed based on the elaborated design models. Although UML is widely used in model-based engineering, it is not suitable for model-based analysis directly due to its lack of formal semantics. Thus, current model-based analysis approaches transform UML models into formal languages dedicated for analyses purpose, which may introduce accidental complexity of implementing the required model transformations.

The recently introduced fUML standard provides a formal semantics of a subset of UML enabling the execution of UML models. In this paper, we show how fUML can be utilized for analyzing UML models directly without having to transform them. We present a reusable framework for performing model-based analyses leveraging execution traces of UML models and integrating UML profiles heretofore unsupported by fUML. A case study in the performance analysis domain is used to illustrate the benefits of our framework.

Categories and Subject Descriptors

I6.4 [**Computing Methodologies**]: Simulation and Modeling; D2.2 [**Software Engineering**]: Design Tools and Techniques; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Design, Performance

Keywords

Model-Based Analysis, fUML, UML Profiles

QoSA 2013 Vancouver, Canada

1. INTRODUCTION

To guarantee that non-functional requirements are fulfilled by a software system, it is of uttermost importance to consider the system's non-functional properties beginning from an early stage of its development. When applying the model-based engineering paradigm in developing software systems, models are used throughout the whole development process serving as the main specification of the system to be developed. Early in the development process, models are defined on a high level of abstraction and continuously refined in the next steps during the development process until an executable system can be derived (semi-)automatically using model transformations and code generation [9]. Thus, model-based engineering emphasizes the importance of abstractions by means of models that act as the primary artifact in the development process. This also facilitates model-based analysis of functional and non-functional properties of the modeled system already in an early development stage to ensure the quality of the system under development and to avoid costly fixes in later phases of the development process.

UML is currently the most adopted general-purpose modeling language in model-based engineering [11], which lead to the emergence of several UML-based analysis approaches for analyzing functional and non-functional properties of a system. However, due to the lack of formal semantics of UML, which is one of its most criticized issues [10, 16], these model-based analysis techniques follow a translational approach, i.e., the UML models have to be translated into different formal (modeling) languages before specific analyses can be performed. For instance, in software performance engineering, many approaches translate UML models with additionally annotated properties into dedicated performance models, such as queuing networks, suited for carrying out performance analyses (cf. Balsamo et al. [2] for a survey).

Translational approaches for model-based analysis have the advantage that existing analysis techniques and tools for the target language can be directly exploited. The major drawback, however, is that an additional level of indirection and complexity is inevitable. Implementing the translation of models from the source language (e.g., UML) into models of the target language used for the analysis is a complex task, requiring deep knowledge not only of the semantics and the metamodels of the source and the target languages, but also of model transformation techniques. Moreover, for several analysis tasks, additional information is necessary, which is, in the domain of UML, annotated using dedicated

UML profiles (e.g., the profiles MARTE [14] and DAM [6]). Thus, the translation also has to correctly incorporate this additional information in the target language. Another challenge in developing translational approaches is translating back the results of the analysis carried out on the model of the target language into the source language. If such a backwards translation is missing, the analysis results may not be comprehensible for the modelers, as they are usually only familiar with the source language. The complexity of translating the analysis results back to the source language lead to the unsatisfactory fact that only very few translational model-based analysis approaches provide the analysis results on the level of the source models (cf. Balsamo et al. [2] pointing this out for the software performance engineering domain). As a result, these approaches do not gain the adoption they deserve outside of academia. Unfortunately, the complex and costly development of the necessary translations (into the formal target language and back again) has to be repeatedly performed for every new analysis method that requires a new combination of target language used for carrying out the analysis and UML profile used for annotating information additionally needed for the analysis, which also hampers significantly the emergence and development of new tools for analyzing different properties of a system that is modeled using UML.

To overcome these issues, France et al. [9] suggested the integration of the analysis algorithms directly with the modeling language used in systems development. Following this suggestion, we propose in this paper such an approach for analyzing a system modeled with UML by utilizing runtime information obtained by executing the models using the virtual machine defined in foundational UML (fUML). Foundational UML [13] is a new standard of the Object Management Group (OMG) that defines the operational semantics of a subset of UML (containing parts of the UML packages *Classes*, *Common Behaviors*, *Activities*, and *Actions*) and provides a virtual machine for executing UML models compliant to this subset. Unfortunately, the fUML virtual machine lacks in providing detailed runtime information about the execution of UML models and ignores UML profiles in the sense that a runtime representation of applied stereotypes and their tagged values is missing in fUML. UML profiles, such as MARTE and DAM, however, are indispensable for analyzing especially non-functional properties of the modeled system, such as for instance performance or reliability, as it is required to annotate additional information in the models (e.g., resource usages, execution times, workload, and failure probabilities) for carrying out the analyses.

In this paper, we address these limitations with the following contributions. First, building upon our previous work on obtaining and representing detailed runtime information about the execution of fUML models [12], we show how this runtime information can be used for performing model-based analyses of the modeled system directly on model level without having to translate the analyzed model into another formal language. Second, we present a framework for bridging the gap between executable fUML models and UML profile applications that are required for model-based analysis using a dedicated integration layer.

The remainder of the paper is organized as follows. Section 2 describes the execution of UML models using the fUML virtual machine, presents how runtime information obtained through the execution can be used for carrying out model-based analyses, and points out current limitations of fUML with respect to analyses. Section 3 presents our proposed model-based analysis framework facilitating model execution based on fUML overcoming the aforementioned limitations. Section 4 illustrates how to exploit our framework with a case study in the domain of model-based performance analysis. Related work is presented in Section 5 and Section 6 concludes the paper providing an outlook on future work.

2. MODEL EXECUTION AND ANALYSIS WITH FUML

In this section, we first provide some background on fUML and the standardized fUML virtual machine for executing models. Second, we outline our extensions of the fUML virtual machine that we presented in previous work [12] to allow for obtaining detailed runtime information about the execution of fUML models. Finally, we present how this runtime information can be used for enabling the model-based analysis of functional and non-functional properties of the modeled system and point out heretofore existing limitations regarding analyses that we address in Section 3.

2.1 Model Execution Based on fUML

Foundational UML (fUML) is a new standard of the standardization body OMG, which defines the operational semantics of a subset of UML and provides a virtual machine for executing UML models compliant to this subset. The fUML subset contains parts of the UML packages *Classes*, *Common Behaviors*, *Activities*, and *Actions*. In essence, fUML enables the execution of UML activities.

Besides a description of the operational semantics, the standard is accompanied by a Java-based reference implementation[1] of an *fUML virtual machine* for executing compliant UML activities. This virtual machine takes as input an fUML-compliant activity to be executed, as well as input values for the activity's input parameters. The fUML activity has to be provided in terms of the in-memory representation (i.e., instantiated Java classes) that is provided by the implementation. After execution, the virtual machine provides the output values of the activity's output parameters as result.

Although the provided virtual machine is realized with a high-quality implementation of the operational semantics of UML defined in the fUML standard and constitutes a major contribution towards the utilization of executable UML models, it has one major drawback: it provides solely output values of the executed activity's output parameters as result of the model execution and does not offer any means for runtime observation and analysis. Thus, except for verifying whether the actual output values correspond to the expected ones, the execution of the fUML activity cannot be reconstructed and examined, which hinders exploiting the full potential of having executable models significantly.

2.2 Extended fUML Virtual Machine

To address the limitations of the standardized fUML virtual machine, we recently introduced a command API as well as an event mechanism into the fUML virtual machine[2]

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

[2] In the following we refer to the fUML virtual machine extended with a command API and an event mechanism pre-

in previous work [12], to allow observing and controlling the execution of activities at runtime.

The *command API* enables to start and stop the execution of an activity, to execute an activity stepwise, as well as to set breakpoints for suspending the activity execution at a particular activity node. The introduced *event mechanism* informs about each change of the runtime state during the execution of an activity. Events are triggered when the execution of an activity starts and ends, when the execution of an activity node of the activity starts and ends, and when the activity execution is suspended. Additionally, events are triggered when extensional values (objects and links) are created, destroyed, and modified.

Whereas the event mechanism enables building tools that react on certain states of the execution, events are not the most appropriate representation for reasoning and analyzing the execution of an activity a posteriori. Therefore, we also build, based on the events raised during the execution, a dedicated *trace model* as additional output of the model execution. This trace provides the information necessary for reasoning about the runtime behavior of the executed model, such as the call hierarchy among executed activities, the chronological execution order of the activity nodes contained in the executed activities, the input provided to and the output produced by the executed activity nodes and activities, as well the token flow among activity nodes and activities. Based on this runtime information about the model execution it is possible to build dedicated analysis tools that analyze the expected behavior of the modeled system. For more information on how the trace model is designed we kindly refer to [12].

2.3 Analysis Based on Model Execution

Being equipped with a trace model that records the execution of an fUML activity and, thereby, reflects the expected behavior of the modeled system, we may now analyze a system's functional properties, such as its functional correctness, by interpreting and checking the execution trace. For instance, the trace can be used to check, if certain activity nodes have been executed in a certain order, if certain activities have been called in course of the execution of a particular activity, and if the correct outputs have been provided by the execution of these called activities constituting intermediate result values of the execution.

For carrying out further analyses of the system using a model-based approach, such as non-functional analysis, additional information has to be annotated which cannot be captured in plain fUML models. For instance, to reason about the resource demands of an activity carried out by a system, the expected execution time, power consumption, amount of allocated memory as well as bytes exchanged over communication networks have to be annotated for all actions contained within the activity as provided by domain experts. Similarly, for reliability analyses, additional information about failure rates of each action can also be required. Such information is usually annotated in the model, using dedicated UML profiles (e.g., the UML profiles MARTE [14] and DAM [6]). However, it is heretofore not possible to take such additional information into account in the analysis when relying on fUML solely, because fUML does not provide a support for utilizing information that is annotated in terms of UML profile applications.

sented in [12] when using the term *extended fUML virtual machine*.

3. A FRAMEWORK FOR MODEL-BASED ANALYSIS WITH FUML

Many UML-based analysis approaches rely on additional information that is annotated in the UML models, most prominently using dedicated UML profiles. To facilitate taking this additional information into account when analyzing UML models based on execution traces produced by the extended fUML virtual machine, we present a novel framework that seamlessly integrates the fUML execution capabilities with UML models and applied profiles. An overview of this framework is depicted in Figure 1 and discussed in the following.

Input.

The framework accepts as input a *UML model* and *profile application(s)* extending this UML model. As our proof-of-concept implementation of the proposed framework is based on EMF, we support UML models that conform to the EMF-based implementation of the UML 2 metamodel. Thus, supported UML 2 models can be developed conveniently, for instance, using the Papyrus[3] UML 2 editor.

UML/fUML Integration: fUML Adapter.

Before we execute the UML model for analysis purposes, we have to provide it in the respective in-memory representation that is dictated by the fUML virtual machine. Moreover, fUML supports only a subset of the complete UML and does not include the ability to carry profile applications. Therefore, we developed a dedicated *fUML adapter*, which truncates all unsupported UML elements, as well as the profile applications, before it adapts the truncated UML model into an executable *fUML view* that corresponds to the in-memory representation of fUML models processable by the fUML virtual machine. As all model elements that are not supported in fUML are truncated, the behavior of the system under consideration should be specified using the subset of UML activities supported by fUML (in contrast to, e.g., UML state machines); otherwise, the parts that are not fUML-compliant cannot be included in the model-based analysis.

fUML-based Model Execution.

The *fUML view* created in the previous step can now be executed using the *extended fUML virtual machine* (cf. Section 2). Utilizing the extensions of the fUML virtual machine, we obtain an *fUML execution trace* from the virtual machine that provides detailed information on the executed parts of the UML activities such as the number of executions of each contained UML action, their input and output values, etc. This detailed trace provides an excellent base for reconstructing the model execution and, based on this information, for analyzing and reasoning about the expected behavior and properties of the system under consideration. However, the fUML virtual machine is not aware of the additional information that has been annotated in the original UML model through profile applications. As a result, also the obtained execution trace only refers to model elements from the fUML view, which do not reflect the additional

[3] `http://www.papyrusuml.org`

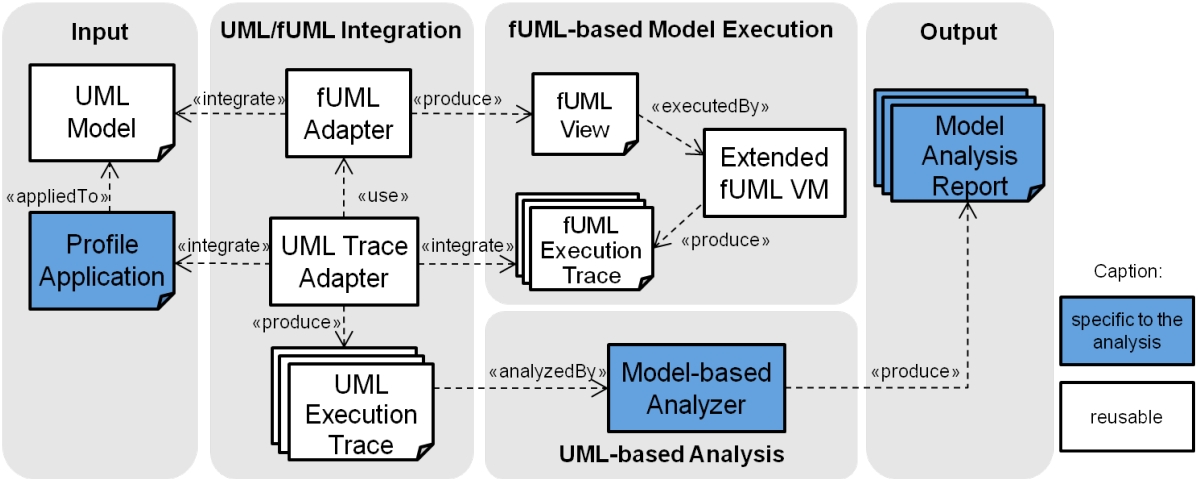


Figure 1: Model-based analysis framework based on fUML.

information captured using profile applications. Therefore, we first have to integrate the original UML model, including the applied profiles, with the plain fUML execution trace as discussed in the following.

UML/fUML Integration: UML Trace Adapter.

The plain fUML execution trace is now adapted back to the original UML model by a dedicated *UML trace adapter*. This trace adapter exploits the mapping information from the aforementioned fUML adapter to map the executed model elements of the fUML view with the original elements of the UML model, as well as the profile that is applied to it. As a result an integrated *UML execution trace* is created, which now refers to the original elements of the UML model instead of the fUML view. This integrated UML execution trace may now serve as the sole source of information for performing the analysis of the modeled system.

UML-based Analysis.

A model-based analysis is typically performed to answer certain questions about the expected behavior of the modeled system. The expected behavior is directly reflected by the trace of the model's execution and can now be analyzed using a *model-based analyzer*, which computes certain metrics by interpreting the execution trace and the additional information captured using profile applications. As the questions to be answered about a system may vary significantly, the actual model-based analyzer, as well as the additional information annotated in the considered models in terms of profile applications, is tailored to specific aspects that should be analyzed. Our framework enables to implement any kind of model-based analysis that is based on the expected behavior of a system and on informations that can be annotated in terms of profiles without dictating any specific analysis methodology. Also the *output* of the analysis in terms of a *model analysis report* may depend heavily on the respective question to be answered. Thus, our framework makes no restrictions regarding which type of report is most appropriate to present the results of an analysis.

In summary, we may conclude that the proposed framework offers the advantages that *(i)* no translation of the UML models used for developing the system into other dedicated analysis models is necessary for carrying out analyses, that *(ii)* the analysis results do not have to be translated back from a target language used for analysis into the source language used for modeling the system (i.e., UML) in order to provide feedback to the modeler because the analyzer can be developed directly based on the representation of the execution trace which refers to the actual UML model, that *(iii)* different analysis techniques and methodologies can be performed on one model by applying multiple profiles to the same model and installing dedicated analyzers for answering specific questions, and that *(iv)* the same model can be used for the analysis of functional and non-functional properties of the modeled system.

4. PERFORMANCE ANALYSIS WITH FUML

In this section, we illustrate how the proposed fUML-based analysis framework presented in Section 3 and depicted in Figure 1 can be suitably tailored to carry out an early estimation of the performance of a component-based software system. For this goal we take inspiration from an existing model-based analysis approach proposed by Smith, namely Software Performance Engineering (SPE) [18].

SPE introduces a domain-specific notation called *execution graphs* (EG), for creating *software execution models*. Execution graphs are platform-independent models which represent the dynamics of software along with its demands for resources. They can be analyzed to provide a static analysis of the mean-, best- and worst-case execution times of the modeled software. EGs are particularly suitable during the early stages of a software development process because an EG

> *"[...] characterizes the resource requirements of proposed software alone, in the absence of other workloads, multiple users or delays due to contention for resources."* [18]

The EG notation has been provided with its own EMF-based metamodel (namely, Software Performance Model Interchange Format, S-PMIF [19]) and analysis tools [17] in order to make it exploitable in model-based development processes for non-functional validation purposes (cf. [5, 7]).

In this section we present how our proposed framework can be utilized for developing a *model-based performance analyzer* that is capable of analyzing suitably annotated UML activities, just as EGs, without the need for model transformations between UML and EG notations. We illustrate the potential benefits of our fUML-based analysis framework through a leading case study in the e-health domain already used in some previous work [4, 5].

The following subsections introduce the case study along with *(i)* its input UML model and performance annotations, *(ii)* the traces obtained from the execution of UML activities, and *(iii)* the performance results obtained by the analysis of these execution traces utilizing our proposed fUML-based analysis framework.

4.1 Input: eHealthSys UML Model

eHealthSys is a distributed component-based system that provides software services to support doctors' everyday activities such as the retrieval of patients' data (*RPD service*) where descriptive texts about the patients' medical histories and diseases, as well as images like X-rays are combined in a multimedia report that is displayed on the doctors' personal digital assistance device (*PDA*).

Both, the software architecture and the hardware platform of *eHealthSys* have been modeled in UML[4]. The additional information required to enable the performance analysis has been annotated through a subset of stereotypes and tagged values from the MARTE profile. The software architecture, the hardware platform, and the performance annotations are separately described in the following.

Software Architecture.

The structural and behavioral specification of the software architecture of *eHealthSys* has been modeled using UML class diagrams and activity diagrams, using only model elements supported by fUML.

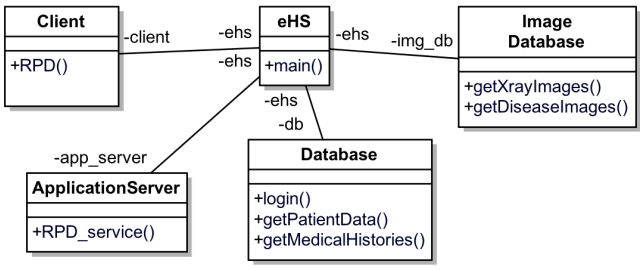


Figure 2: *eHealthSys* software architecture.

eHealthSys includes four different types of software components depicted in the class diagram shown in Figure 2. A *Client* is the front-end that doctors use to invoke the *RPD_service* running on an *Application Server* that can access two different databases, one for textual data (*Database*) and one for multimedia files (*Image Database*).

UML activities have been used both for creating the instances of the *eHealthSys* software components (Figure 3(a)) and for modeling the behavior of their operations (Figure 3(b)).

In particular, the *eHealthSys main* activity depicted in Figure 3(a) creates a single instance of each software component (*create object actions*) and sets up the values of their properties (*add structural feature value actions*) before starting the execution scenario by invoking the operation *RPD* (*call operation action*) on the *Client* instance.

The behavior of the operation *RPD* is specified by the UML activity depicted on the left side of Figure 3(b). It first invokes the remote operation *RPD_service* on the *Application Server* and then elaborates and displays the resulting multimedia report locally (*call behavior action*) on the doctor's PDA. For sake of illustration, all the operations involved in the provision of the operation *RPD_service* on the *Application Server* are visualized in the sequence diagram depicted on the right side of Figure 3(b). Each lifeline corresponds to an object created in the *main* activity. The sequence diagram shows the remote operation calls among the participating software components. Since UML interactions (visualized, e.g., using sequence diagrams) are not part of the fUML standard, a dashed line surrounds the sequence diagram indicating that it is not considered by the model-based performance analysis. However, the behavior of the respective operations is also specified using UML activities compliant to fUML to enable their consideration in the model-based performance analysis.

Hardware Platform.

The *eHealthSys* UML model also includes the modeling of the *eHealthSys* hardware platform. It is composed of a set of execution hosts, where the *eHealthSys* software components are deployed on, and details the computing, communication, and storage resources exploitable by the deployed software components.

An excerpt of the *eHealthSys* hardware platform is shown in Figure 4 where the software components depicted in Figure 2 are allocated on their respective execution hosts. This is modeled through the *abstraction relationships* pointing from the software components (*Client*, *Application Server*, *Database*, *Image Database*) to the execution hosts (*PDA*, *AppServer Host*, *DB Host*, *ImageDB Host*) where they are deployed on.

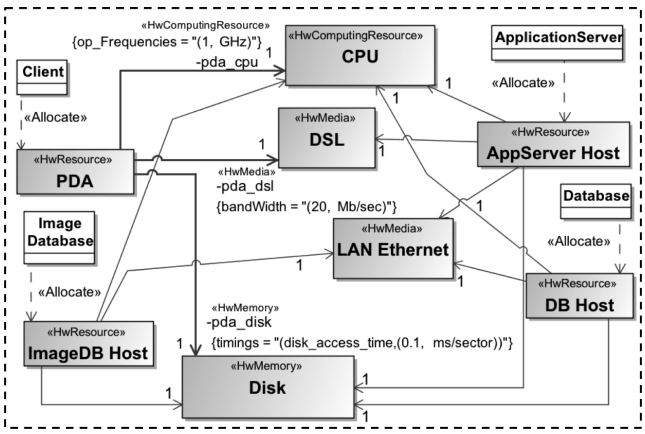
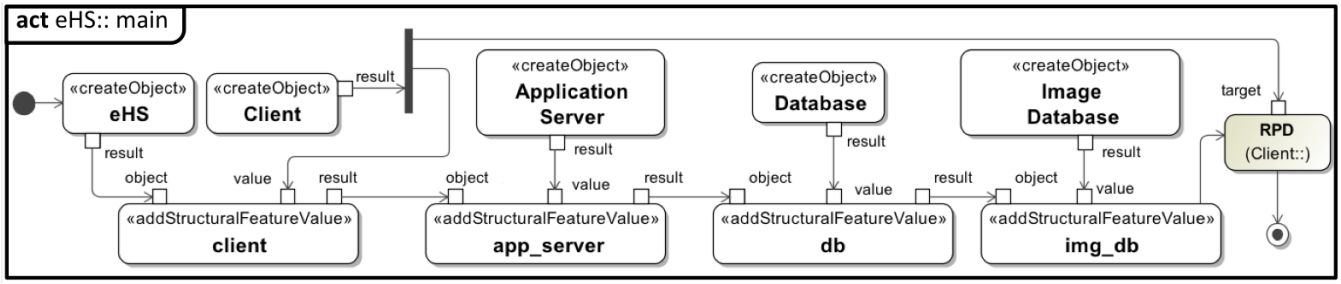


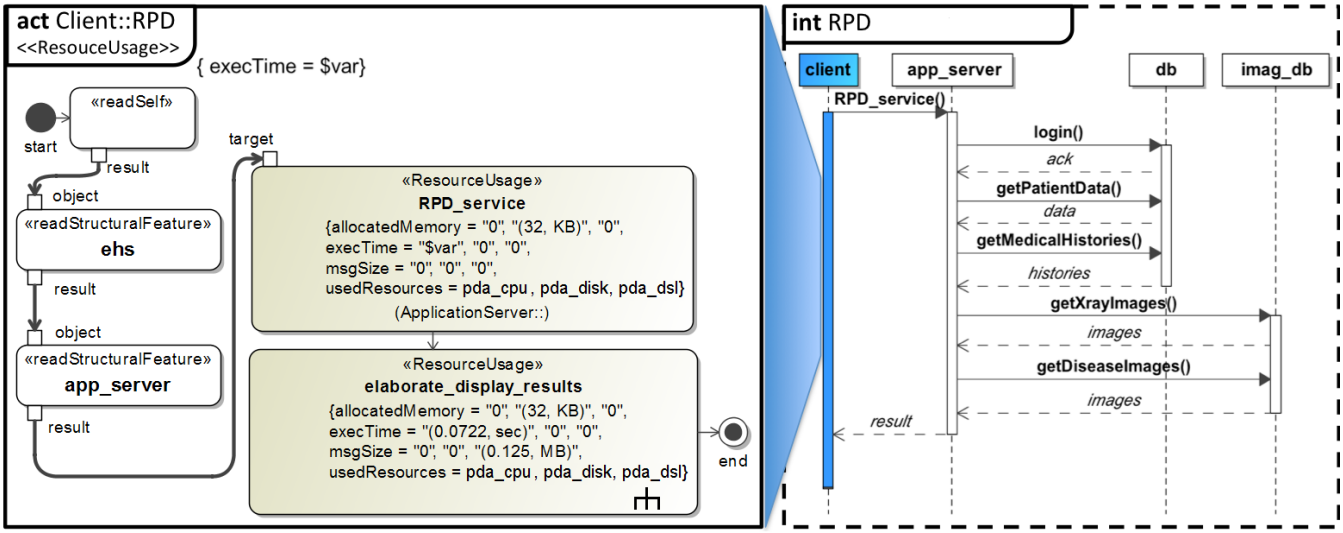
Figure 4: An excerpt of the *eHealthSys* hardware platform.

The execution hosts are equipped with their own processing units (*CPU*), storage (*Disk*), and communications resources (*DSL*, *LAN Ethernet*). Each hardware configuration is represented by *directed associations* from the execution hosts to their hardware resources.

[4]The *eHealthSys* UML model can be downloaded at `http://www.modelexecution.org/`.



(a) Instantiation of the *eHealthSys* software components according to its software architecture.



(b) Behavior specification of the *eHealthSys* software components.

Figure 3: Software behavior specification of the *eHealthSys*.

It is worth noting that we deliberately chose the UML package *Classes* (i.e., the modeling elements *class*, *association*, and *relationship*) instead of the UML package *Deployments* for modeling the *eHealthSys* hardware platform. This choice follows the guidelines suggested by MARTE for modeling hardware platforms [14]. Applying these guidelines has the additional advantage of fitting the modeling restrictions imposed by fUML that does not support UML *Deployments*. However, it has to be noted that the classes specifying the hardware platform (i.e., the gray-colored classes in Figure 4) are not taking part in the model execution carried out in the course of the performance analysis, because no behavioral specifications are associated to them; only the behavior of the software components are specified and analyzed (cf. Figure 2).

Performance Annotations.

The *eHealthSys* UML model described so far needs to be properly extended to make it suitable for performance analysis. In particular, the model is lacking details about *(i)* the hardware resources of the execution hosts, as well as *(ii)* their utilization by the allocated software components. This information can be introduced by means of the MARTE profile.

MARTE is standardized by OMG and supports the specification, design, and verification/validation stages in those model-based approaches that adopt UML as their main design notation.

Table 1 lists a small subset of the stereotypes defined by MARTE along with their names, some tagged values, and the UML elements where they have been applied in the *eHealthSys* model in accordance with the modeling requirements of our case study.

Table 1: MARTE stereotypes applied to the *eHealthSys* model.

Stereotype Name	Applied to	Tagged Values	View
Allocate	Abstraction	-	HW
HwComputingResource	Class, AssociationEnd	op_Frequencies	HW
HwMedia	Class, AssociationEnd	bandWidth	HW
HwMemory	Class, AssociationEnd	timings	HW
HwResource	Class	-	HW
ResourceUsage	Activity, CallOperationAction, CallBehaviorAction	allocatedMemory, execTime, msgSize, usedResources	SW

Among them, the stereotype *ResourceUsage* has been applied on the behavioral units of the RPD service (i.e., on the specified activities and their contained actions) to annotate an estimation of the amounts of hardware resources required to execute them, as shown in the activity diagram depicted in Figure 3(b).

Table 2: Execution times of the behavioral units of the *eHealthSys* RPD service.

Component	Behavioral Unit	Model Element	Remote/Local	execTime	Input/Calculated
Client	RPD	Activity	-	185,7795	calculated
	RPD_service	CallOperationAction	remote	185,7073	calculated
	elaborate_display_results	CallBehaviorAction	local	0,0722	input
Application Server	RPD_service	Activity	-	185,7073	calculated
	login	CallOperationAction	remote	0,0722	calculated
	getPatientData	CallOperationAction	remote	0,2888	calculated
	getMedicalHistories	CallOperationAction	remote	2,3087	calculated
	getXrayImages	CallOperationAction	remote	18,5570	calculated
	getDiseaseImages	CallOperationAction	remote	9,2473	calculated
	elaborate_send_results	CallBehaviorAction	local	155,2333	input
Database	login	Activity	-	0,0722	calculated
	elaborate_send_results	CallBehaviorAction	local	0,0722	input
	getPatientData	Activity	-	0,2888	calculated
	elaborate_send_results	CallBehaviorAction	local	0,2888	input
	getMedicalHistories	Activity	-	2,3087	calculated
	elaborate_send_results	CallBehaviorAction	local	2,3087	input
Image Database	getXrayImages	Activity	-	18,5570	calculated
	elaborate_send_results	CallBehaviorAction	local	18,5570	input
	getDiseaseImages	Activity	-	9,2473	calculated
	elaborate_send_results	CallBehaviorAction	local	9,2473	input

In particular the following tagged values of the stereotype *ResourceUsage* have been used in the *eHealthSys* model:

- *usedResources* identifies the set of resources used by the particular execution scenario. These resources have to be chosen from the processing, communication and storage resources equipping the execution host where the corresponding software component is allocated (e.g., *pda_cpu*, *pda_disk*, *pda_dsl*, respectively, for the *Client* component deployed on the *PDA*, see Figure 4);
- *allocatedMemory* represents the amount of memory that is demanded to execute the behavioral unit (e.g., the required amount of RAM or hard disk space);
- *msgSize* corresponds to the amount of data transmitted by a resource (e.g., via a network card installed on the *PDA*);
- *execTime* represents the time required to execute a behavioral unit.

Table 2 lists the execution times needed for executing the behavioral units of the RPD service, along with the software component providing each unit, the name of the unit, the UML model element used to represent the unit, and the information whether the respective unit is executed locally or remotely invoked.

We provide an estimation for these execution times of the behavioral units of the RPD service that reflects the characteristics of the underlying processing, storage and communication resources. These characteristics are also annotated in the UML model on the classes representing the hardware resources through the respective MARTE stereotypes.

Figure 4 partially shows the characteristics of the hardware resources of *eHealthSys* which we assumed for our case study. For example we suppose that a doctor's *PDA* is equipped with a 1GHz *CPU* (*pda_cpu*), a solid state *Disk* (*pda_disk*) with an access time of 1 ms per sector, and communicates through a DSL network (*pda_dsl*) with a bandwidth of 20 Mb\sec. Similar hardware configurations have been defined for all the execution hosts of *eHealthSys* and annotated on the UML model using MARTE.

The estimated execution times needed for executing the behavioral units of the RPD service have been annotated in the UML activities using the tagged value *execTime* of the stereotype *ResourceUsage* in accordance with the allocation of the software components on the execution hosts (cf. Figure 4). These stereotype applications are partially depicted in Figure 3. It is worth noting that only the execution times of locally executed behavioral units have to be directly annotated by the modeler on the input *eHealthSys* UML model. The remaining execution times are calculated by the performance analyzer as described in the next subsection (cf. last column of Table 2 for the information which execution times have been annotated in the model as input and which have been calculated by the analyzer). In Figure 3(b) we adopt a $-prefixed notation as suggested by MARTE for model variables that, in this case, are used as placeholders for the execution times which will be calculated by the performance analyzer described in the following.

4.2 Software Performance Analysis with fUML

Based on the annotated *eHealthSys* UML model described in Section 4.1, we are able to calculate the execution time of the RPD software service. For this we implemented a model-based performance analyzer leveraging our proposed fUML-based analysis framework introduced in Section 3.

The performance analyzer implements a graph reduction algorithm for calculating the end-to-end execution time of the RPD service. The used algorithm was originally introduced by Smith [18] for calculating the execution time of a software service based on EGs.

Thanks to the simplifying assumptions underlying a software modeling notation like EGs (no workloads, no multiple users, no delays), an early performance analysis can be conducted by simulating a small *eHealthSys* instance model composed by the five running objects as instantiated by the activity depicted in Figure 3(a). Indeed there is no need to create multiple instances of clients and application servers as well as to count the execution of actions.

EGs are composed of nodes which represent the processing steps that perform a function of the modeled software as well as of arcs representing the execution order of the nodes. Nodes can either be basic or expanded nodes representing atomic behavioral units or subordinate processing steps, respectively. The latter are then further detailed by a subgraph which can again consist of basic and expanded nodes. The algorithm for calculating the execution time of a system based on EGs [18] starts the calculation at basic nodes of the considered EG and sums up the annotated execution times of sequentially ordered basic nodes. This sum constitutes the execution time of the subgraph containing the nodes. If this subgraph details an expanded node, this sum constitutes the execution time needed for performing the expanded node and is propagated to the superordinate subgraph containing the expanded node. This algorithm continues until the whole EG has been considered. The resulting value constitutes the execution time needed for performing the software service modeled in the considered EG.

For applying this algorithm defined for EGs on UML activities, the modeling concepts used in EGs, i.e., basic and expanded nodes, have to be mapped to modeling elements available in UML activities. In our case study, basic nodes are mapped to *call behavior actions* (these are the locally executed behavioral units listed in Table 2), whereas expanded nodes are mapped to *call operation actions* (representing remotely invoked behavioral units listed in Table 2). Applying this mapping of UML activities to EGs, the algorithm for calculating the execution time on the basis of UML activities works as follows: It starts at *call behavior actions* and sums up the annotated execution times of sequentially ordered *call behavior actions*. This sum constitutes the execution time of the activity containing the considered *call behavior actions*. This value is propagated up to the calling activity, i.e., the calculated sum constitutes the execution time of the *call operation action* calling the respective activity.

The performance analyzer implemented in this case study leverages our proposed fUML-based analysis framework for performing the calculation of the execution time of the RPD service. The *eHealthSys* UML model is provided as input to the framework which subsequently executes the model utilizing the extended fUML virtual machine and provides the UML execution trace to the performance analyzer for carrying out the calculation.

An excerpt of the UML execution trace for the RPD service is depicted in Figure 5. It shows that the three activities specifying the operations *eHS::main()*, *Client::RPD()*, and *ApplicationServer::RPD_service()* were executed. This information is represented in the trace by the *Activity Execution* objects ae1, ae2, and ae3 respectively. The trace also holds the information, that the activity *eHS::main()* called the activity *Client::RPD()* which again called *ApplicationServer::RPD_service()* by executing *call operation actions* which is represented by the *Call Action Execution* objects c1 and c2. In the course of the execution of *Client::RPD()*, also one *call behavior action* was executed (c3) calling *elaborate_display_results*. In the course of the execution of *ApplicationServer::RPD_service()*, the operations of the *Database* and *Image Database* were called by executing *call operation actions* (c4 to c8), as well as the behavior *elaborate_send_results* by executing one *call behavior action* (c9).

For calculating the execution time of the RPD service based on this UML execution trace, the analysis algorithm first sums up the execution times annotated for the executed actions contained in the activity specifying *ApplicationServer::RPD_service()*. This sum (185,7073 seconds in this example) constitutes the execution time of the activity and is propagated to the action calling this activity (its execution is represented by the object c2 in the trace). The execution time of the activity for *Client::RPD()* is calculated again by summing up the execution times of the executed contained actions (c2, c3). The execution time of the action execution represented by c2 is the propagated value calculated for the execution time of the activity for *ApplicationServer::RPD_service()*. The action execution represented by c3 constitutes the execution of the *call behavior action elaborate_display_results*, thus, the value for its execution time is directly annotated in the UML model. This sum built for the execution time of *Client::RPD()* (185,7795 seconds) is again propagated to the action calling the activity *Client::RPD()* (c1) and constitutes the execution time of the top-level activity *eHS::main()* which is the end-to-end execution time of the RPD service provided by the *eHealthSys*. Please note that the execution times of the actions represented by the objects c4 to c8 in the trace are calculated using the same algorithm.

The performance analyzer provides the analysis results in two formats. For each carried out performance analysis a spreadsheet is created listing all considered behavioral units (i.e. *call actions* and *activities*) of the *eHealthSys* input model with their (given or calculated) execution times. Additionally, a copy of the input UML model is created and the stereotype *ResourceUsage* is applied to each behavioral unit capturing the respective execution time in the tagged value *execTime*.

This straightforward mapping of the performance analysis algorithm for EG to UML activities can certainly be implemented by transformational analysis tools and be part of a more complex tool chain supporting more complex performance analysis approaches as shown in [17]. However, by building on the extended fUML virtual machine and UML execution traces, we avoid transformations of the UML model to an external notation (e.g., EG) and simplify the model analysis and the supporting tool chain as well.

The current analysis algorithm implemented by the presented performance analyzer is still subject to certain limitations due to its prototypical nature:

- Our prototype does not cover the whole original EG notation that includes, among all, also loop and conditional nodes. Despite the fact that their mappings to UML is also straightforward, the UML model would need to consider the execution probabilities of alternative control flows that are not yet taken into account during the model execution carried out by the fUML virtual machine.

- Further, the algorithm assumes that the behavioral units (i.e., actions and activities) are executed sequentially. For parallel paths a simple worst-case analysis is carried out.

Implementing more advanced analysis algorithms using the presented analysis framework based on fUML is subject to future work as discussed in Section 6. However, we showed that a model-based analyzer can be easily implemented by utilizing our fUML-based analysis framework

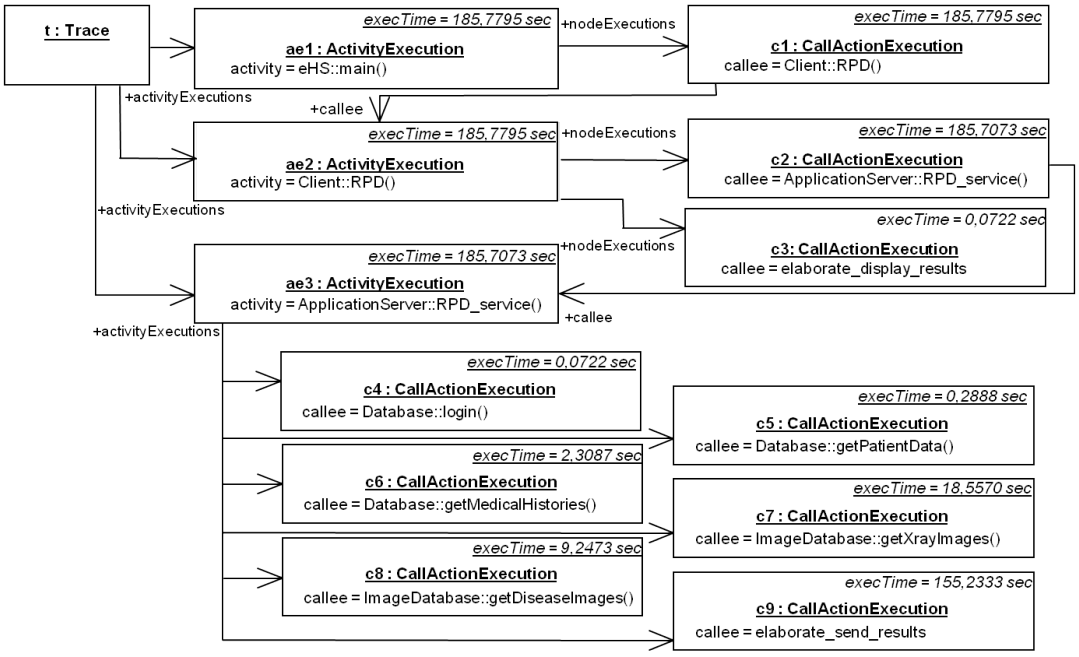


Figure 5: Excerpt of the UML execution trace provided by the fUML-based analysis framework to the performance analyzer.

leveraging UML execution traces obtained by executing the models on the extended fUML virtual machine.

5. RELATED WORK

The fUML standard has been published in 2011. To the best of our knowledge, no research papers have been published so far that investigate the combination of fUML and UML profiles. However, even without an explicit support for the latter, there are several ongoing research efforts towards the exploitation of fUML in MDE approaches for verification and validation activities.

Romero and Ferreira propose an MDA-based approach applied to the domain of space real-time software for sake of code generation and schedulability analysis [15]. In their approach platform independent models (PIM) and test cases are specified using fUML activities. Non-functional properties are annotated on the activities using the UML profile MARTE. However, for carrying out the schedulability analysis, fUML activities are translated into AADL models [8].

Benyahia et al. [3] evaluate how well the current fUML semantics supports the formalization of concurrent and temporal semantic aspects which is required for the design and analysis of real-time embedded systems. They illustrate how the standard fUML execution model, as well as the fUML virtual machine, have to be extended for this purpose to explicitly incorporate a *scheduler* into fUML that, at each step of the model execution, determines the activity node to be executed next according to certain scheduling policies (e.g., first-in-first-out (FIFO)).

The same limitation has been addressed by Abdelhalim et al. [1]. In contrast to Benyahia et al. [3], they do not propose an extension of the standard fUML execution model but rather present a model-based framework that translates fUML activities into communicating sequential processes (CSP) for performing a deadlock analysis detecting possible scenarios leading to deadlocks which are provided as UML sequence diagrams. Their framework is implemented as a plug-in for MagicDraw[5], a commercial UML tool that also provides a plug-in for executing fUML models.

All aforementioned approaches do not utilize the combination of fUML and UML profiles for carrying out model-based analyses. In the approach proposed by Romero and Ferreira [15], non-functional properties are annotated using the MARTE profile but, as stated by the authors themselves, they are ignored at the PIM level and are only considered when fUML-based PIMs are transformed into AADL models for sake of schedulability analysis (which is not feasible at the moment in fUML as stated in [3]). The approaches presented in [1] and [3] do not make use of UML profiles at all. However, in [3] the authors express their interests and efforts in providing a methodological and tooled framework that exploits UML profiles to deal with semantic variations of fUML and to support different model analyses directly in fUML (e.g., schedulability analysis).

Our framework pursues a similar goal, namely integrating analysis methods directly with UML by utilizing model execution. The expected benefits of directly utilizing the execution of UML models for carrying out model-based analysis are twofold: *(i)* the costly translation of UML models into formal languages dedicated to specific analysis purposes is avoided and, hence, *(ii)* the implementation and maintenance of supporting analysis tool sets is eased significantly. With our fUML-based approach presented in this paper, we offer both of these benefits and showcased them by developing a performance analyzer that implements an analysis method originally based on execution graphs (EGs) directly on UML models suitably annotated with MARTE profile applications. A translation of UML models into EG models can now be omitted and it is not necessary to use additional

[5] `http://www.magicdraw.com`

external tools for analyzing EGs in UML-based performance approaches (as it is necessary in [5] and [7]).

Despite its current limited support for more complex model-based performance analysis (such as for instance presented in [3]) our case study illustrates how fUML *as is* can be combined with UML profiles for analysis purpose and shows that our proposed framework is powerful enough to be adopted in early model-based validation activities of software systems.

6. CONCLUSIONS

In this paper, we showed how execution traces obtained from executing fUML models can be used for validating non-functional properties of a system that is modeled using UML. As for the validation of non-functional properties additional information that is annotated in terms of UML profiles, such as the MARTE profile, is required, we proposed a framework that integrates fUML execution traces with UML profile applications. Based on this framework, users may develop model-based analysis tools more easily without any restrictions to a certain analysis methodology and without having to translate UML models to other formal models, such as queuing networks, and back again. To validate our proposal, we developed and presented a model-based analyzer for validating the expected execution time of a modeled system.

In future work, we plan to develop further model-based analysis tools and apply different analysis methodologies based on the proposed framework. To also support methodologies that build upon probabilities annotated in the UML model, we will extend our framework to allow for obtaining all possible execution traces from a UML model and weight the values in the traces, such as execution time, according to the annotated probabilities in each trace. Moreover, we will work towards supporting parallelism in UML activities and additional parts of UML that are currently not integrated with fUML, such as state machines.

7. REFERENCES

[1] I. Abdelhalim, S. Schneider, and H. Treharne. An integrated framework for checking the behaviour of fUML models using CSP. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2012.

[2] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.

[3] 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 *Proceedings of the IFIP Conference on Distributed and Parallel Embedded Systems (DIPES'10)*, pages 43–54. Springer, 2010.

[4] L. Berardinelli, V. Cortellessa, and A. Di Marco. A unified approach to model non-functional properties of mobile context-aware software. In *Proceedings of the International Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NFPinDSM'09) at MODELS*, 2009.

[5] L. Berardinelli, V. Cortellessa, and A. Di Marco. Performance modeling and analysis of context-aware mobile software systems. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'10)*, pages 353–367, 2010.

[6] S. Bernardi, J. Merseguer, and D. C. Petriu. A dependability profile within MARTE. *Software and Systems Modeling*, 10(3):313–336, 2011.

[7] V. Cortellessa and R. Mirandola. PRIMA-UML: A performance validation incremental methodology on early UML diagrams. *Sci. Comput. Program.*, 44(1):101–129, 2002.

[8] P. Feiler, D. Gluch, and J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, CMU/SEI-2006-TN-011, DTIC Document, 2006.

[9] R. B. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of the Workshop on the Future of Software Engineering (FOSE'07) at ICSE*, pages 37–54, 2007.

[10] B. Henderson-Sellers. UML–the good, the bad or the ugly? Perspectives from a panel of experts. *Software and Systems Modeling*, 4(1):4–13, 2005.

[11] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the International Conference on Software Engineering (ICSE'2011)*, pages 471–480. ACM, 2011.

[12] T. Mayerhofer, P. Langer, and G. Kappel. A runtime model for fUML. In *Proceedings of the International Workshop on Models@run.time (MRT'12) at MODELS*, 2012.

[13] Object Management Group. Semantics of a foundational subset for executable UML models (fUML), version 1.0, February 2011.

[14] Object Management Group. UML profile for MARTE: Modeling and analysis of real-time embedded systems, version 1.1, June 2011.

[15] A. Romero and M. Ferreira. An approach to model-driven architecture applied to space real-time software. In *Proceedings of the International Conference on Space Operations*, 2012.

[16] B. Selic. The less well known UML. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2012.

[17] C. Smith, C. Lladó, V. Cortellessa, A. Marco, and L. Williams. From UML models to software performance results: An SPE process based on XML interchange formats. In *Proceedings of the International Workshop on Software and Performance (WOSP'05)*, pages 87–98. ACM, 2005.

[18] C. Smith and L. Williams. *Performance solutions: A practical guide to creating responsive, scalable software*, volume 1. Addison-Wesley Boston, 2002.

[19] C. U. Smith, C. M. Lladó, and R. Puigjaner. Performance model interchange format (PMIF 2): A comprehensive approach to queueing network model interoperability. *Perform. Eval.*, 67(7):548–568, 2010.