Integrating Software Models and Platform Models for Performance Analysis

Vittorio Cortellessa, Pierluigi Pierini, and Daniele Rossi

Abstract—System performance is a key factor to take into account throughout the software life cycle of modern computer systems, mostly due to their typical characteristics such as distributed deployment, code mobility, and platform heterogeneity. An open challenge in this direction is to integrate the performance validation as a transparent and efficient activity in the system development process. Several methodologies have been proposed to automate the transformation of software/hardware models into performance models. In this paper, we do not take a transformational approach; rather, we present a framework to integrate a software model with a platform model in order to build a performance model. Performance indices are obtained from simulation of the resulting performance model. Our framework provides a library of predefined resource models, model annotation and integration procedures, and simulation support that makes the performance analysis a much easier activity. We present the results obtained from two different industrial case studies that show the maturity and the stability of our approach.

Index Terms—Software performance, software model, platform model, UML, simulation.

1 Introduction

One of the issues that prevents the performance validation from being embedded as common practice in the software life cycle is the distance between the worldview adopted by software developers and performance experts. Software developers usually describe a system through static and dynamic models that deal mostly with functional aspects. Performance experts are interested in additional nonfunctional aspects, such as the operational profile (i.e., the estimation of execution probabilities of different software subsystems), and need to integrate software models with characteristics of target platforms in order to devise meaningful performance models. Ad hoc performance models have been built in the past (usually based on Queueing Networks and Stochastic Petri Nets) to estimate the performance of specific software/hardware systems. However, there are several reasons, such as short time to market, required expertise, etc., that prevent the building of ad hoc performance models, and lack of performance validation often causes large software projects to fail [10], [17].

A real breakthrough in the performance validation of software systems was experienced less than 10 years ago [28] with the introduction of new approaches to the problem solution. The rationale behind this new trend was that software designers do not need to know details about performance analysis; they only need to observe the analysis results to improve their design. Thus, the transparency of the process becomes a key factor in making performance analysis practices appealing to the software engineering community. The paradigm of all these new techniques can be summarized as follows: Annotate software models with performance data, integrate annotated models with (estimated or available) platform characteristics, and translate the extended models into performance models. As long as annotations can be embedded within usual software artifacts, developers do not have to change their practices. Extended models contain all the data necessary to build performance models, so model transformation techniques can be applied to automatically generate performance models.

In the last few years, several methodologies have been introduced to annotate software models (e.g., UML models, Use Case Maps, etc.) and transform the annotated models into performance models (e.g., Petri Nets and Queueing Networks). A quite comprehensive survey of recent work in this field can be found in [3]. Fig. 1 illustrates the basic steps (i.e., the paradigm) that these methodologies share. Square boxes represent inputs and outputs of the performance analysis process, ovals represent intermediate artifacts, and rounded dashed boxes represent the software environments where the artifacts are typically produced. A new challenge in this direction is to build integrated environments where software developers can

1. build software models,
2. integrate software models with platform models,
3. annotate models with data related to performance,
4. transform annotated models into performance models, and, finally,
5. solve the performance models to obtain the indices of interest.

A few environments have been recently introduced that offer developers support for some of the five previous steps [3], [24]. However, no environment is yet available to support an integrated approach to the performance validation along the software life cycle.

- *V. Cortellessa and D. Rossi are with the Dipartimento di Informatica, Università dell'Aquila, Via Vetoio, 1-67010, Coppito (AQ)-Italy. E-mail: cortelle@di.univaq.it.*
- *P. Pierini is with Technolabs SpA, SS 17, Località Boschetto-67100, L'Aquila (AQ)-Italy. E-mail: pierluigi.pierini@technolabs.it.*

Manuscript received 29 Oct. 2005; revised 9 Aug. 2006; accepted 15 Mar. 2007; published online 28 Mar. 2007.
Recommended for acceptance by A. Wellings.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0286-1005.
Digital Object Identifier no. 10.1109/TSE.2007.1014.

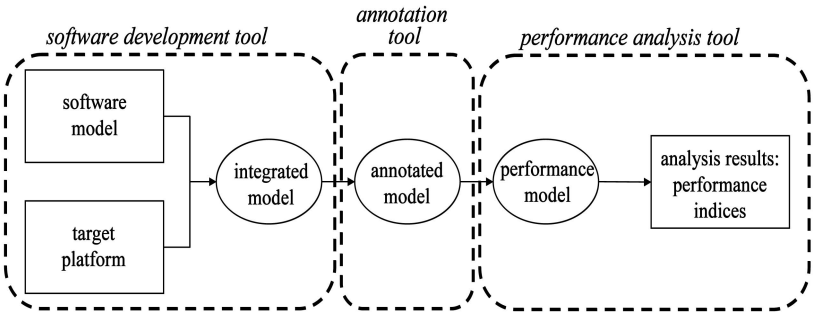

Fig. 1. A new paradigm for the performance analysis process.

In this paper, we introduce a framework that enables integrating a software model with a platform model, and thereafter annotating the integrated model with data related to performance (e.g., the operational profile, the amounts and types of resources requested). The framework provides a library of predefined resource models to assemble platform models, facilities for model annotation, software and platform model integration procedures, and support for the integrated model simulation. Besides, we present a methodology to build and solve a performance model within our framework.

The model simulation is a key aspect to eliminate the need to transform an integrated model into a performance model (such as a Queueing Network) to compute performance indices, thus overcoming inaccuracies that might be introduced in the transformation process. In addition, the whole performance process illustrated in Fig. 1 can be executed in an unique environment.

The framework and the methodology that we describe in this paper are independent of the languages and tools adopted, and they can be instantiated in different settings. However, we illustrate the approach via the specific case of the UML-RT notation, which is an UML extension for real time systems [21]. In practice, as a consequence of choosing UML, any UML tool that allows simulation of UML diagrams can be used as a front end support to our methodology. In addition, since the major UML-RT features are part of the UML 2 standard, our approach can be instantiated in UML 2 as is, without any extensions. This is in fact one of our future goals, as illustrated in the conclusions.

The framework obviously suffers from a typical simulation drawback, which is the long elapsed time to get reliable results. On the other hand, it exploits the potential of modeling complex systems that cannot be treated by analytical models. In addition, a unique modeling and analysis environment keeps out of the picture any simulation code detail and allows us to focus on the performance assessment.

We faced two main problems in obtaining this result: 1) modeling the platform resources to assemble a platform model, and 2) devising a standard way to annotate a model and to represent the interactions between software and platform, with the aim to be as "transparent" as possible to the software development process. Our transparency concept can be reformulated as straightforward traceability. Being "transparent" (in our approach) does not mean that the software model remains unchanged, but that the latter is not "heavily" modified, for example by moving some transition in statecharts or some connector in a Capsule Diagram. All of the annotations and integrations that we perform can be straightforwardly traced back by erasing the codes and additional capsules that we introduce. Therefore, the original software model can be considered, in practice, as untouched.

The support provided by the IBM-Rational Rose RealTime (RRT) tool set [30] has allowed to instantiate our framework on the UML-RT notation and to experiment our solutions in a real software modeling and analysis environment.

The goal of this paper is to illustrate our approach and show how it simplifies the software performance analysis process. We clearly state the pros and cons of this approach and we compare it to the major existing methodologies. This is the first comprehensive study that we have introduced in this area as it puts in the same picture ideas and results that we have presented in our previous work [1], [6], [7], as well as the application of the approach to real world case studies. An extensive description of the whole project, with implementation details, can be found in [8].

The paper is organized as follows: In Section 2, we compare the related work with our approach. Section 3 illustrates all the steps of our approach. In Section 4, we show the results obtained on two real-world case studies; in Section 5, we sum up the lessons learned from this experience. Finally, in Section 6, we provide concluding remarks and future perspectives.

2 Related Work

In the first part of this section, we introduce related work on simulation models for performance analysis that does not depend on the UML notation. Thereafter, we present recent results in the functional and nonfunctional validation of UML models based on simulation, and we point readers interested in analytical approaches to the survey in [3].

In [22], the idea of integrating performance prediction into a software design environment was first presented. The work is based on the ObjecTime environment that enables execution of a design model. The model execution is used in this approach to produce software execution traces (i.e., *angio traces*) that, in turn, allow the automated generation of a performance model based on Layered Queuing Networks [16]. The model is integrated with resource data that come from a Resource Function Management Utility (RFMU). The role and capabilities of an RFMU have been detailed in [23], where experimental results have also been presented. In practice, an RFMU has three main functions: 1) measuring functions on given hardware platforms and operating systems, 2) storing measured data in a repository, and 3) retrieving resource demand values corresponding to actions in the software model generated from angio traces.

The main difference between the above approach and the work presented in this paper is that the prototypes in

our library are generic enough in their behaviors that they can be reused (by just modifying some parameters) to build new platform architectures, whereas the resource demands in RFMU are preevaluated for specific pairs $\langle platform, operating\_system \rangle$.

An interesting view on the integration of performance in SDL/MSC-based systems has been given in [13]. In particular, the QUEST approach is described in this work, which is based on the adjunction of time consuming machines that model the congestion of processes due to limited resources.

In our approach, we export this idea to a wide domain of models. Our solution provides a general methodology to build platform models and to integrate them with software models. We also devise an implementation of the methodology within the UML domain, which is the current standard in software modeling.

Concerning the simulation engine that we adopt in our implementation, we note that many design and simulation environments for performance analysis are today available (e.g., HyPerformix [27]). However, each environment requires that a software model is built in a specific notation, and this can be a heavy limitation for integrating those environments in the software lifecycle practices. Given that our approach is based on UML, the above comments on the breadth of its scope also apply here.

Let us survey recent simulation work for performance analysis based on UML models.

In [15], a tool prototype has been proposed that allows the simulation of UML Sequence Diagrams. The diagrams are animated as event traces. A similar approach has been introduced in [2], where the SimML simulation framework is used to generate simulation code from UML Class and Sequence Diagrams annotated with stochastic data. In [14], UML Class and State Diagrams are simulated to verify functional properties of software models. However, all these approaches are not equipped for performance assessment, in that the simulations only provide a mean to observe the system dynamics (i.e., without collecting any performance index).

In [4], data related to the performance are annotated on UML Use Case, Activity and Deployment Diagrams, following the syntax introduced in [29]. An annotated model is then translated into a simulation code whose execution produces values of performance indices that, in turn, can be annotated back on the original diagrams. This approach is, however, based on model transformation, whereas we integrate in the same notation a platform model and a software model to be simulated.

In [25], the capability of introducing additional code on states and transitions of UML-RT state diagrams has been exploited to build a framework for verification of timing constraints in real-time systems. Our framework certainly shows larger capabilities because the analysis that we can conduct is not limited to timing constraints but spans over other performance indices (such as utilization, throughput, etc.).

In [16], the approach named Layered Queueing Model (LQM) was introduced to model performance of software/hardware systems; since then, it has been greatly improved and extended [31]. Layered Queueing Networks (LQNs), that is, the notation LQM lies on, is the closest existing notation to the one that we use here to illustrate our approach, that is, UML-RT. However, there is a significant difference between LQNs and UML-RT: An LQN is specifically designed for performance analysis and evaluation, so it also contains blocks that represent the resources. A set of resources can be attached to every component in order to represent the resources that the component requires. This is missing in an UML-RT diagram, which does not have the potential to be used (as is) for performance goals. In this paper, besides introducing a thorough methodology for integrating software and platform models for performance assessment, we illustrate how the UML-RT notation can gain this potential. This UML-RT added capability may have a larger impact on real software development projects since UML-RT is a widely used notation for modeling software/hardware systems in many domains, whereas LQM is adopted only in the performance context.

In [18], the MASCOT design language has been integrated with time and resource representations to build simulation models for system performance analysis. This approach is also somewhat similar to ours. However, as remarked above, our approach introduces a thorough methodology to build and solve simulation models, and, in addition, the implementation that we provide here works on the more widely adopted UML-RT notation.

With the work proposed in this paper we intend to contribute to the research issues in the field of performance and real-time design that have been nicely summarized in [19] as follows: 1) characterization of (real-time) platforms, 2) understanding the relationship between software and its platform, and 3) evolution of contention analysis techniques.

We contribute to the first issue with the introduction of a library of classes (here modeled in UML-RT) that represent the structure and the behavior of platform resources (e.g., CPUs, LANs, etc.). Hence, the integration of a software and platform model can be obtained by appropriately introducing in a software model (e.g., based on UML-RT) the set of resource classes that represent the target platform. The support of a simulation environment (for purposes of illustration, we use RRT here) can be exploited to solve the performance model and obtain performance values.

We contribute to the second issue by characterizing the relationship between software model and running platform as follows. The integration of software and platform models is based on two types of additional data: 1) the platform topology and the mapping of software components to platform sites and 2) the resource requests from software components. We provide a description of these data later in the paper. However, both types of data have been considered key factors in [19] for a standardization of the platform concept.

As a contribution to the third issue, a novelty of our approach is that we simulate UML models without translating them into simulation code (as in [4], for example). Our approach allows for integration of the software and the platform models in the same environment where the software has been modeled (e.g., RRT), and (if possible) the same tool is used to run the model simulation. Hence, this is the first approach for performance validation that does not require transformation of models or generation of code.

3 A Framework to Integrate a Software Model with a Platform Model

The methodology described in this section can be implemented using any notation with the following basic features: static and dynamic description support for basic elements (such as

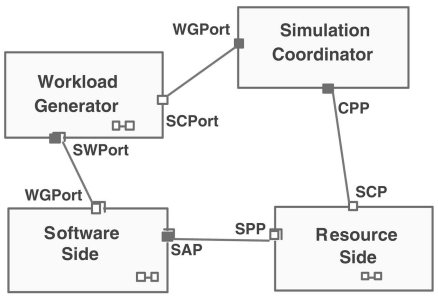

Fig. 2. Software and Platform sides of an UML-RT integrated model.

software components), capability of model annotation, and support for model simulation. In this paper, we illustrate our methodology using the UML-RT notation [21].[1]

The idea of integrating a software model with platform specifications for performance validation goals can be accepted by software developers only if the integration does not bring changes to their development practices. In other words, transparency is a key factor for such an approach, and it is one of the major achievements of the framework that we describe in this section.

The simulation of a software model may allow for checking functional aspects, such as correct event sequencing, deadlock-freedom, and liveness. The simulation of an integrated software/platform model allows us to simultaneously take into account the software dynamics and the platform mechanisms that generate critical latency time in the software execution, mostly due to the resource contention. By interconnecting software and platform models, we provide the capability for each step in the software model to fire transitions in the platform model that simulate the waiting and consumption time of resources needed to perform the step. From an MDA perspective [12], a software model represents a Platform Independent Model, and no actual performance indices can be obtained from it, whereas the model obtained upon integration with a platform model is a Platform Specific Model whose simulation can provide real performance values to reason about.

A thorough performance validation process, which starts from an existing software model and ends up with the simulation of an integrated model, includes the following basic steps:

1. representing basic platform resources,
2. assembling a platform model,
3. integrating software and platform models, and
4. Running the simulation and analyzing results.

In what follows, we also refer to the software model as *software side* and the platform model as *resource side*, with the intent of remarking that they represent two sides of the same integrated model. In addition, the term *processing node* indicates a node with processing capabilities in a distribute environment.

In Fig. 2, we show a high-level view of an UML-RT integrated model, ready for simulation. The software side capsule represents the original software model as built from software developers, only consisting of interacting software components (i.e., the software architecture). The resource side capsule represents the model of a target platform where the system shall run. Resource requests travel unidirectionally from the software model to the platform model over the connector(s) between these two high-level capsules. The additional two capsules included in the diagram, namely the simulation coordinator and the workload generators, are required, respectively, to 1) start, terminate, and manage the simulation and 2) model the stochastic behavior of the external input to the software system (i.e., provide the software side with the appropriate workloads that allow for studying the performance indices of interest). While the former component is predefined in the framework that we propose, the latter one is closely related to the modeled system since the workload characterization heavily depends on the application domain. Moreover, the design of a workload generator must comply with the interface defined by the simulation coordinator, including the mechanisms required to control the start and the end of the simulation activity.

The framework that we propose is based on the library *PAlib* of components, rules, and scripts that implement such rules in the RRT environment. *PAlib* supports, respectively, the four basic steps of our approach as follows:

1. **Resource prototypes** are modeled and collected in *PAlib* to provide reusable basic blocks for platform models. *PAlib* is currently populated with prototyped models of some of the most used resource types like CPU, mass memory, and network. A resource model is obtained by instantiating and specializing a prototype from the library. The prototypes also embed the probes required to collect and report performance data on a per resource basis. The library is easily extensible with new resource prototypes.
2. **Platform assembly** is supported by guidelines and scripts to build each single processing node and the whole platform model. Additional dispatching components are provided in *PAlib* to standardize the management of resource service requests. Dispatching components also contain the probes required to collect and report performance data on a request basis.
3. **Integration** between the software and resource side is based on annotations that allow the software model to formulate resource requests. A standard protocol enables a single request to ask for multiple resource services necessary to complete an application step. Note that the granularity of a step is not fixed, in that requests can be associated to transitions representing any amount of software logics.
4. **Running the simulation**, interpreting the performance results and giving insights about the software and platform architectures of the system, is the final step of our approach. The modeling environment should provide automation support to compile and simulate the integrated model. During the simulation, a set of log files can be created by the *PAlib* activated probes, thus providing the performance results to be interpreted by the designers.

A detailed description of each step is given in the remainder of this section. Due to our UML-RT implementation, some references to UML-RT can be found in the description, without the loss of generality.

Most of the examples introduced in the following discussion refer to the "client-server video streaming system" (CSVSS) presented in [29]. The system is made of

1. For the sake of readability, we do not provide any introductory concept of UML-RT, and we assume readers are familiar with UML-RT basics.

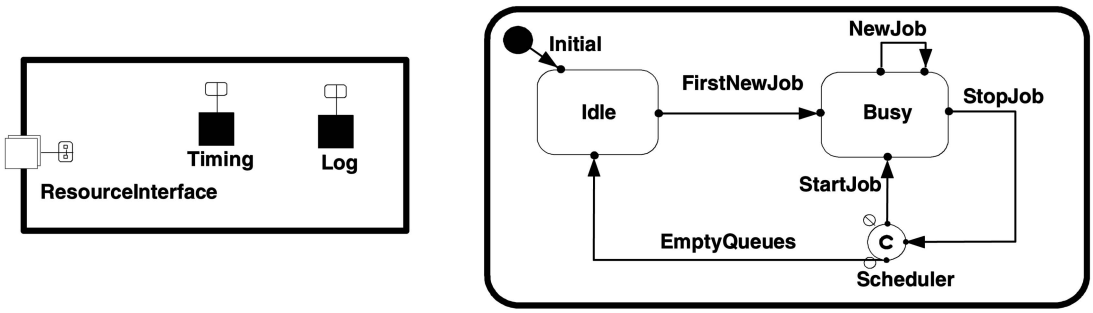

Fig. 3. Generic capsule and statechart of a resource.

five main software components and provides a single service of video streaming along the network. A user performs a video selection through a browser and asks the remote web server to send the video to its site. Based on the user selection, the web server chooses the best video server, the latter initializes a video player on the user site and sends a stream of video frames. The frames are shown to the user through a video window. The main performance goal in this example is the minimization of the response time for each user request.

3.1 Resource Prototypes

We provide a set of prototypes representing types of resources. Each prototype has been implemented as an UML-RT capsule associated with a statechart, which describe, respectively, the structure and the behavior of that type of resource. We have partitioned the types of resources into three main categories: CPUs, mass memories, and physical connectors. Based on this categorization, we have built a library of prototypes so that each prototype can be reused and instantiated within different platform models [8]. Obviously, the number of categories, as well as the number of prototypes within each category, are not fixed; rather, they may be extended by considering and modeling other types of resources. The library extensions allow for building ever more sophisticated hardware platforms, as will be evident in Section 4.

Note that we are not proposing any UML extension here as the class corresponding to a capsule of a certain prototype can be considered as a *PResource* class of the performance analysis domain modeled in the UML Profile for Schedulability, Performance, and Time [29].

3.1.1 Implementing Resources in UML-RT

Fig. 3 depicts a generic resource in terms of its structure (i.e., the capsule) and its behavior (i.e., the statechart). The capsule shows an external port through which it receives the resource request messages. In the capsule are also represented a timer, used to simulate the time consumed by the resource to satisfy a request, and a log port, which allows for logging of the data collected by the resource probes during the simulation.

The resource behavior is described by a statechart, in which two basic states and a decision point are represented: 1) the "*Idle*" state is active when the resource is not in use, 2) the "*Busy*" state when it is engaged in some activity, and 3) the "*Scheduler*" decision implements the specific scheduling and priority policy. Once a demand enters a resource capsule, a set of elementary jobs required to satisfy the request is enqueued (e.g., a disk reading can be partitioned as a set of block reading jobs). When the scheduler selects a job for execution, the capsule moves to a busy state for a specific time. This transition simulates the time spent by the physical resource to execute the job. When all the jobs related to a demand have been processed, a *request satisfied* message is replied back.

Following this general behavior, in our framework we do not explicitly model resources such as locks and semaphores. A certain type of RAM is the only passive resource that we have considered, as modeled in Section 4.1 of this paper.[2]

Later in this section, we analyze how this basic model may be customized for a certain type of resource. For the sake of readability, in this section, we introduce only one type of resource, that is, the CPU, although the library currently contains several other types of resources. In fact, the scope of this paper is not to be exhaustive on the implementation of resources in UML-RT but, rather, to introduce our general approach and illustrate a possible implementation within the UML-RT domain.

Values related to performance can be collected during the simulation on a resource basis, such as the number of arrival and completion requests, the total simulation time, and the total busy time. From these values, it is then possible to calculate performance indices such as throughput, utilization, arrival rate and average response time [11]. More elaborated statistical data can be also collected, such as: 1) the minimum and maximum time spent to process a single request (i.e., *minPeak* and *maxPeak*); 2) resource-specific indices, such as the maximum dimension of the input and output message buffers on a LAN.

3.1.2 Example of a CPU Prototype

In Fig. 4, a CPU with a Round-Robin scheduling strategy is shown. The internal timer is used to define the quantum time assigned to each activated job. The CPU behavior has been modeled with the associated statechart shown in Fig. 4. The CPU leaves the idle state when the first job enters the resource. Two events may occur while being in the busy state. If a new job enters the resource, then it is simply queued. If the CPU quantum expires for the currently processed job, then two alternatives are possible: 1) If the job has been fully processed, either the next job is extracted from the queue (if any) or the CPU goes back to an idle state. 2) If the job still needs processing time, it is queued again and the next job is extracted from the queue and processed.

2. However, the effort to create prototypes for passive resources should not be great, but it is part of library enhancement that has not been necessary until now.

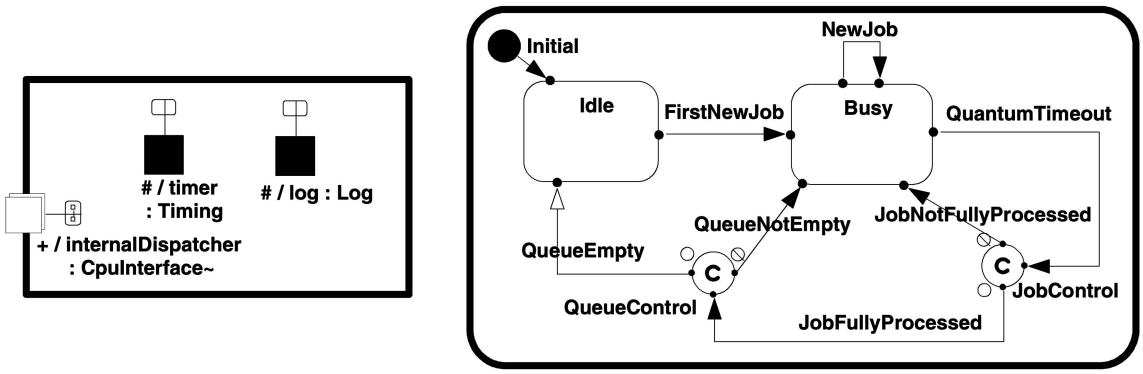

Fig. 4. Capsule and statechart of a Round-Robin CPU.

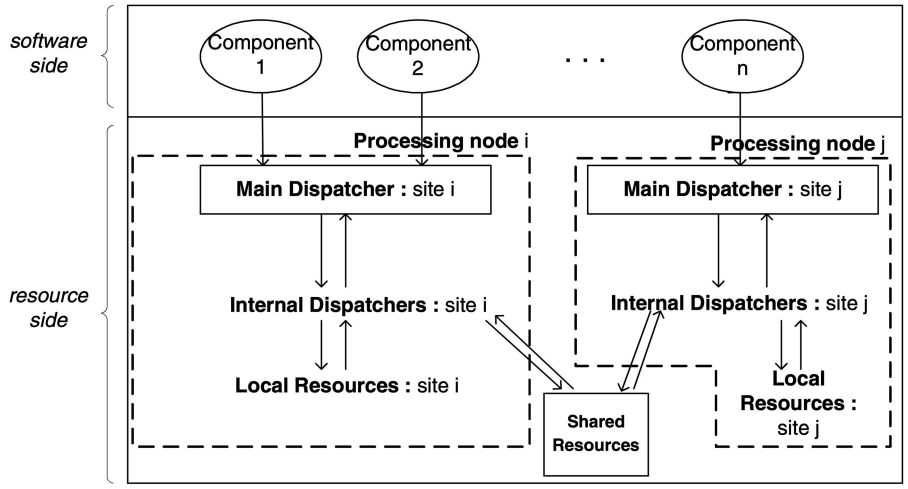

Fig. 5. General architecture of software and resource sides.

3.1.3 Effort to Build a New Resource Prototype

The effort to create new types of resource models through specialization of the basic model presented above is mostly related to 1) the specialization of the resource request interface that defines the specific resource parameters and access methods, 2) the scheduling policy, and 3) additional probes for resource specific performance parameters. After all, from our experience, building a new type of resource does not require a large effort, with the exception of low level and tool specific coding problems that might require, in some cases, significant testing and debugging.

3.2 Assembling a Platform Model

In general, the idea of platform is not a well and formally defined concept. Here, we basically adopt the general definition proposed in [20]. For our purposes, a platform is a processing system that can be partitioned into processing nodes. Each processing node consists of physical resources and a system environment offering a set of services to the hosted software applications (e.g., a PC, a workstation or even a partition of a large-sized machine), and it can be modeled by a processing node embedding a set of private (local) resource instances plus additional supporting components. Note that the structure of a platform does not necessarily represent the complete processing environment, but it may be limited to the resources that play a critical role in the performance analysis.

Fig. 5 represents a general architecture of software and resource sides. Depending on the platform characteristics, one or more processing nodes have to be represented in the model, in addition to the resources shared between processing nodes (e.g., the typical case of a network connection).

A processing node is modeled as a capsule with a three-layer structure. The bottommost layer contains the instances of the local resources (such as a Round-Robin CPU, a Hard Disk, etc.) in addition to connections to shared resources. The uppermost and the intermediate layers represent a sort of middleware that works to manage and dispatch the resource requests from the software side to the physical resources.

The uppermost layer is based on the *Main Dispatcher* component that provides a single access point to send resource request messages from software side to the specific processing node. All the software components hosted on the same site send their resource requests to the same *Main Dispatcher*. We assume that each request originates from a software action that may be performance critical and that it is encoded as a vector made of elementary demands. Each elementary demand represents the amount of a resource category that the action needs to be executed. The *Main Dispatcher* component, following its own strategy, dispatches each elementary demand to the *Internal Dispatcher* that manages the corresponding category of resources in

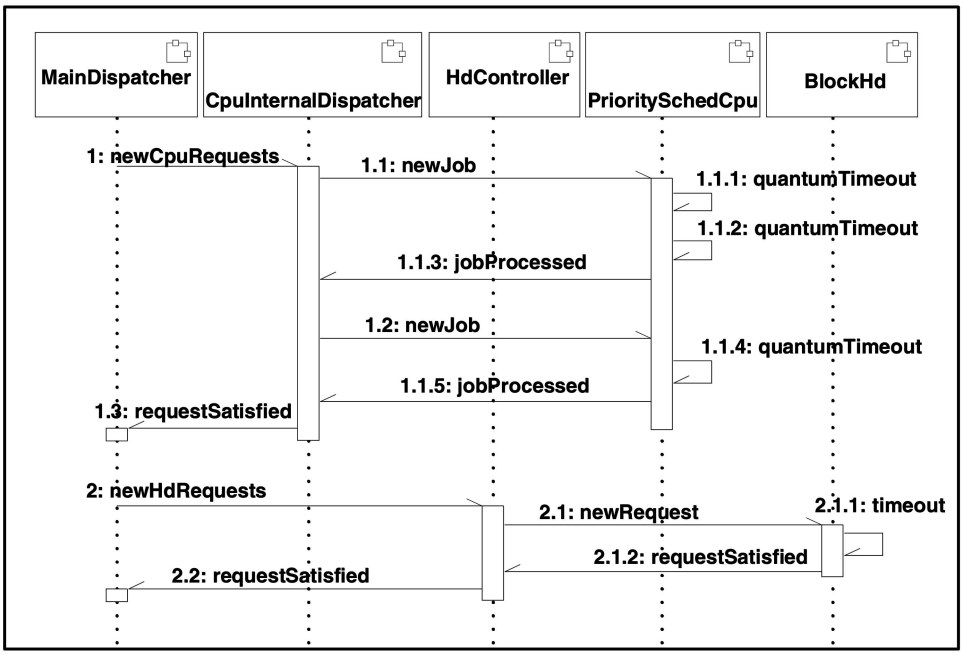

Fig. 6. Dynamics of request dispatching and satisfying.

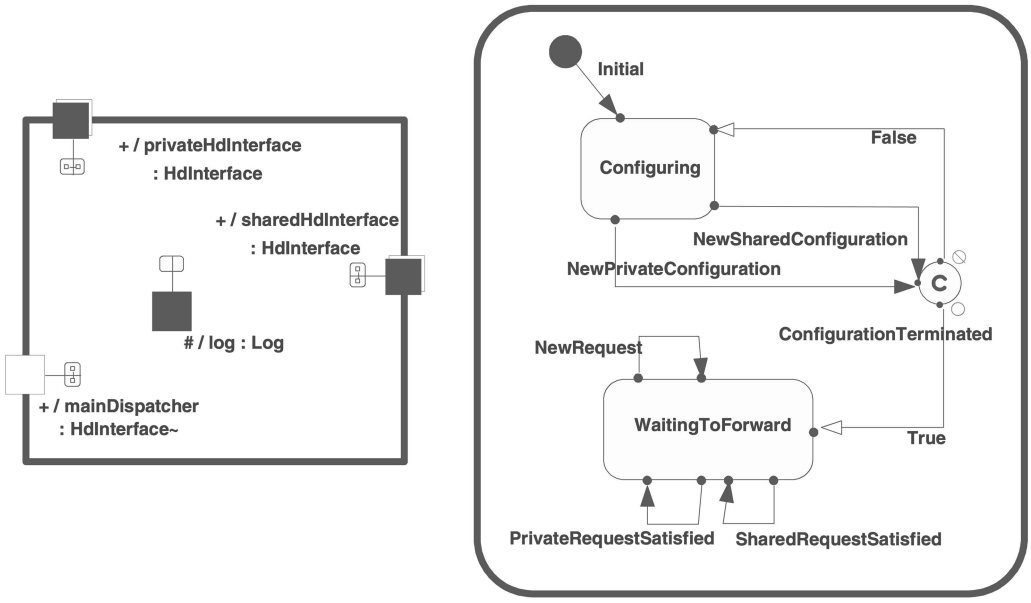

Fig. 7. Capsule and statechart of an *Internal Dispatcher*.

the intermediate layer. Every *Internal Dispatcher*, in turn, following its own strategy, forwards each elementary demand to one of the resource instances that it manages. Thereafter, a "demand satisfied" message is replied back from the resource, through the *Internal Dispatcher*, to the *Main Dispatcher*. Once all the elementary demands making up a resource request have been satisfied, the *Main Dispatcher* updates counters and data required for performance evaluation on a per request basis. The UML Sequence Diagram in Fig. 6 shows such message exchange mechanism.

The statistical data that can be collected per request is usually related to the total execution time of the request itself. Thus, assuming that a request is made of several elementary requests, the total execution time is the time for completing the processing of all the elementary requests.

3.2.1 *An* Internal Dispatcher *for Each Resource Category*

In Fig. 7, we show the capsule and the statechart of an *Internal Dispatcher* (for mass memory resources). It has three external ports: *mainDispatcher*, which is connected to the *Main Dispatcher*; *privateHdInterface* (a multiple port), which is connected to the set of local resources (i.e., those hosted on the considered site; and *sharedHdInterface* (a multiple port), which is connected to the set of resources shared with components hosted on different sites.

After a configuration phase, an *Internal Dispatcher* enters the *WaitingToForward* state. It remains in this state until the

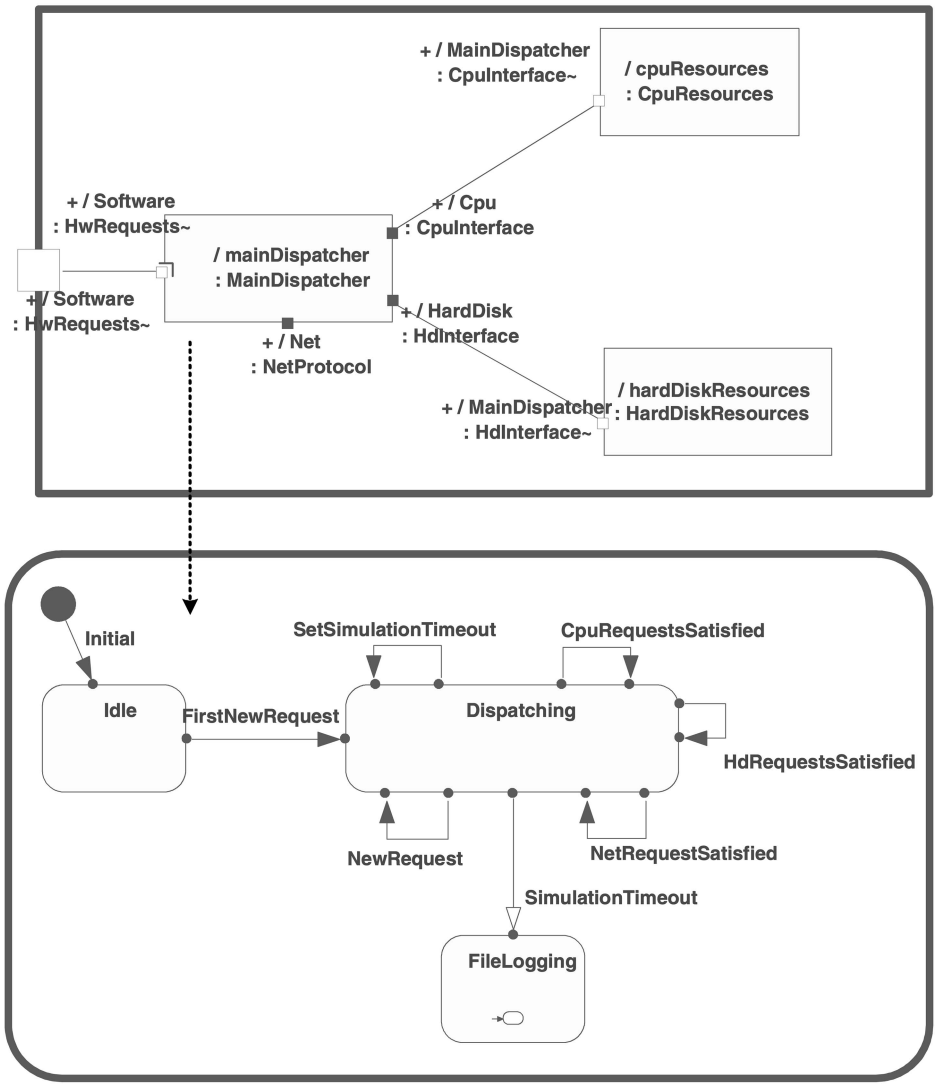

Fig. 8. Capsule and statechart of a *Main Dispatcher*.

end of the simulation, waiting for three self-transitions to be activated: *NewRequest*, which occurs when an elementary demand is forwarded to one of the resource instances, following an internal strategy; *PrivateRequestSatisfied*, which originates a message of "request satisfied" to the *Main Dispatcher*; and *SharedRequestSatisfied*, which originates a message to the *Main Dispatcher* as well.

3.2.2 Main Dispatcher

The main functionality of the *Main Dispatcher* is to expose a single interface to collect composite resource requests from the software side. The requests received are registered and, then, split into elementary demands, each one forwarded to the proper resource category (e.g., CPUs, mass memories) through the *Internal Dispatchers*. The physical resources reply back to the *Main Dispatcher* when their elementary requests are satisfied; thus, the relevant statistics on the request basis can be updated.

The bottom portion of Fig. 8 shows the statechart of a *Main Dispatcher*. From an initial idle state, the *Main Dispatcher* migrates to the *Dispatching* state upon the arrival of the first resource request. It remains in this state until the *Simulation Timeout* expires (i.e., a message from the simulation control is received). This event triggers the transition to the *File Logging* state, where all the simulation statistics are collected and sent out as simulation results.

3.2.3 Effort to Assembly a Platform Model

The assembly of a platform model is very simple, as we have defined specific rules to guide this task. The major effort consists in defining the platform and processing nodes structure (as shown in Fig. 3), and including in each processing node a *Main Dispatcher*, the necessary number of *Internal Dispatchers*, and the resources instances. Thereafter, connections between conjugate ports must be created, thus modeling the relevant communication channels. Finally, shared resources can be added and their interface ports must be connected to the public ports exposed by the related *Internal Dispatchers*.

In our UML-RT implementation, the platform assembly rules have been coded as part of a more general RRT scripting support that saves the designers from heavy editing.

3.3 Integrating Software and Platform Models

In Fig. 5, we have shown the integration of software and resource sides. All of the software components hosted on the same site send their resource requests to the same

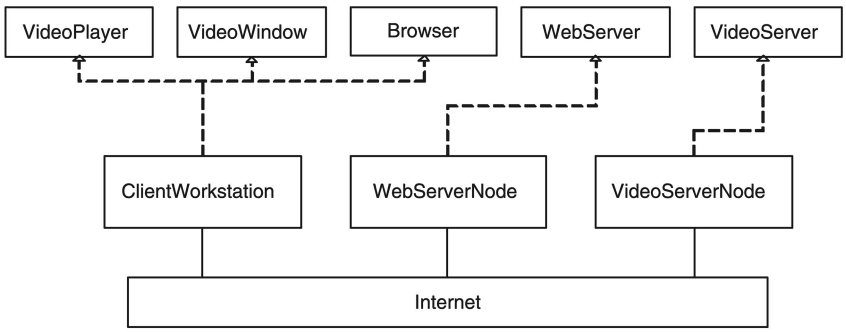

Fig. 9. Deployment for the CSVSS example.

processing node's *Main Dispatcher*. We recall that each request is encoded as a vector of elementary demands. Each elementary demand represents the amount of a resource category that the action needs to be performed.

The integration task consists of the following steps:

1. mapping software components to platform sites (i.e., processing nodes),
2. identifying software components with actions that may be performance-critical,
3. adding to each identified software components the port and connector required to interface to the related processing node,
4. estimating the amount of resource(s) that each identified action requires to be executed, and adding the code required to formulate the resource(s) request.

The first step can be considered as an usual design activity and therefore does not require any relevant additional effort in practice. The second step is somewhat complex and requires experience to be carefully executed. The components identified in this step are the only ones with actions that spend some time in the resource side of the model. A wrong choice of components may result in either overestimating or underestimating the system performance. The effort of the third step is negligible, as it only defines a communication channel between two capsules through a consistent pair of connected and conjugated ports. The last step involves the "annotation" of the software side [8].

Scripting support is available for the third and the last steps to lighten the effort of the integration activity. Scripts allow designers to ignore the annotation mechanisms. However, script efficiency and flexibility depends on the support provided by the modeling tool. In our implementation, we have taken advantage of the RRT capability to create a window-based GUI. Using our scripts, a designer can 1) add an annotation in any point of the original model simply indicating the type and the amount of the required resource(s), 2) remove any included annotation.

As mentioned above, different platform alternatives can be evaluated under the same software model, and different sets of components can be selected in the second step to stress different resources.

As a simple example, in Fig. 9, we show an UML Deployment Diagram that defines a software architecture for the CSVSS example and a platform structure on which the software components may run.

Fig. 10 shows the software side architecture of the CSVSS system in terms of software components and their connections.

In Fig. 11, the software side has been integrated with a resource side (on the bottom of the figure), which is made of three processing nodes (i.e., *clientWorkstationResources*, *wsWorkstationResources*, and *vsResources*). A straightforward mapping of software components to platform sites is also devised. An additional capsule represents the only shared resource among processing nodes, that is the *internet* WAN connection.

Note that the Deployment Diagram represents a good starting point for the designer to build the resource side model. However, the structure of the resource side model is left to the designer on the basis of her/his experience in resource representation and on performance aspects to be analyzed.

We can parameterize this model with resource request amounts, times and workloads. Model parameters may be estimated, measured or assumed (see Section 3.3.1).

In Table 1, we report example values for the main model parameters. In Fig. 12, we also show these parameters as annotations on an UML Sequence Diagram of the system scenario, following the syntax in [29]. Even though Fig. 12 is not functional to the scope of this paper, we intend to remark that we are not introducing any UML extension for our goal; rather, we are using the existing UML stereotypes in [29] to annotate our models.

Note that in case of composite resource requests, where multiple types of resources are requested by a single software action, the *Main Dispatcher* decides the order of resource consumption. It has been shown, in Queueing Networks theory [11], that the order of consumption does not affect the (mean values and variances) of performance indices; thus, whatever criterion is coded in a *Main Dispatcher* will be correct.

3.3.1 Estimating Model Parameters

This activity consists of assigning values to model parameters. The latter ones can be partitioned into three categories: platform characteristics, amount of resources needed, and intensity of workload. The estimation techniques suitable for all the parameter categories obviously depend on the software lifecycle phase in which the system is being modeled.

In good approximation, we can roughly distinguish two scenarios with this respect: 1) running system (i.e., it has been deployed) and 2) system under development (i.e., it is not deployed yet). However, the goal of the performance analysis can be substantially different in the two cases. The analysis of a system under development is aimed at validating if the system performance falls into a range compatible with system specifications. The analysis of a deployed system is a two-step task: First, the model is built

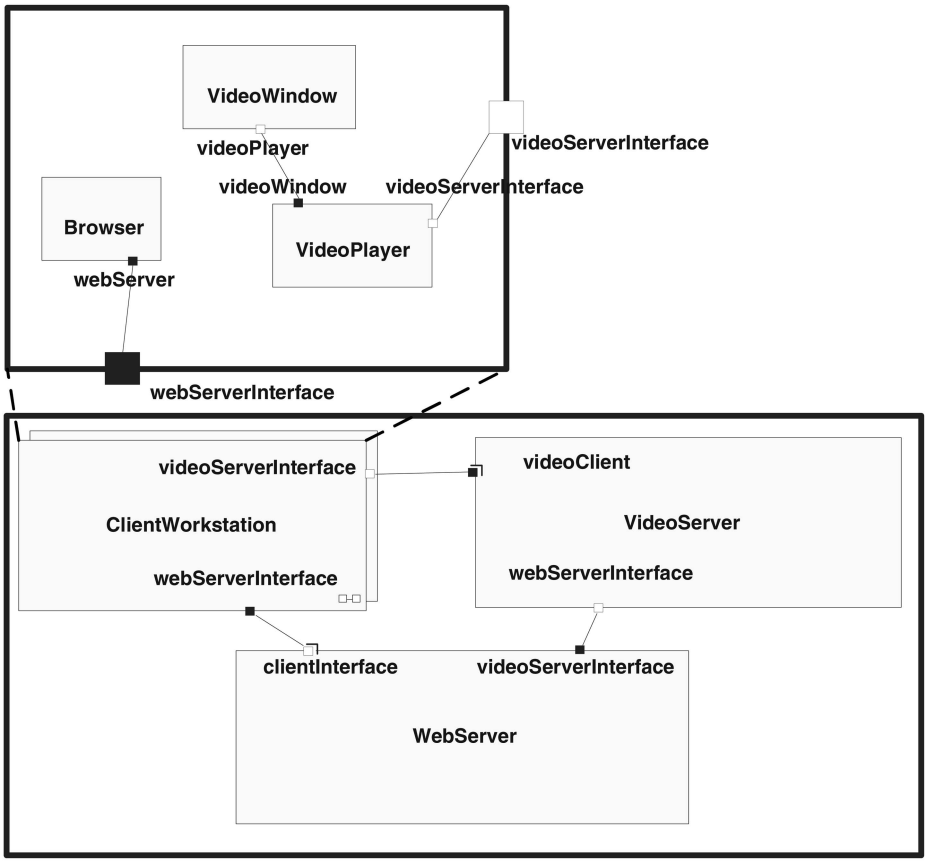

Fig. 10. Structure of CSVSS software side.

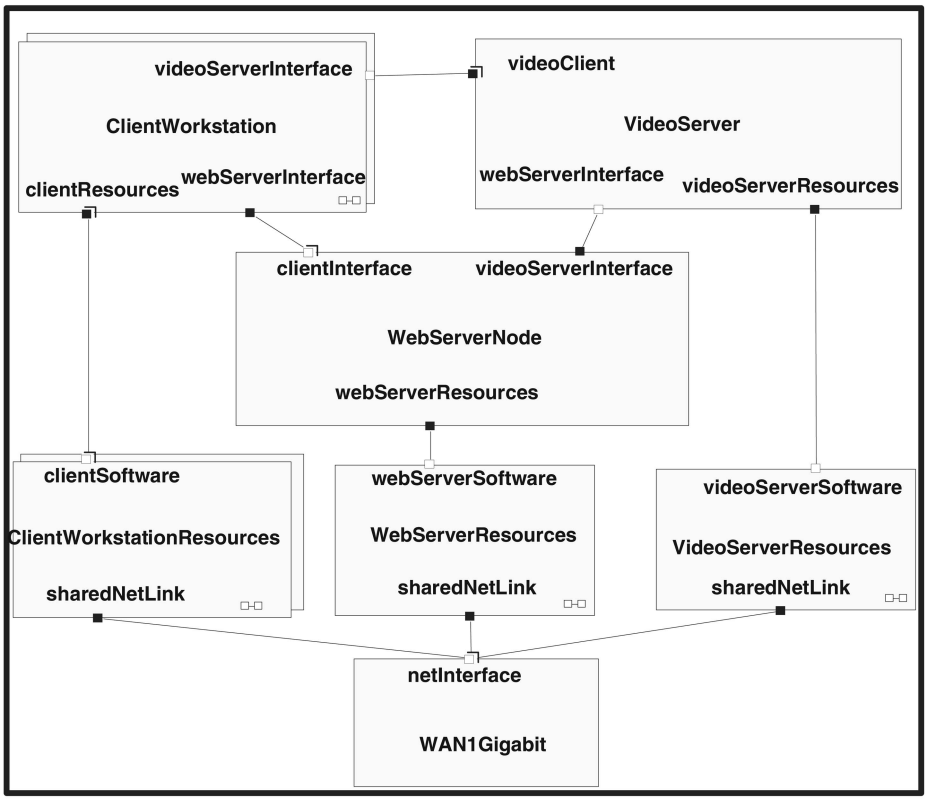

Fig. 11. Integrated modeling of CSVSS software and resource sides.

TABLE 1
The CSVSS Parameters

processing time of video server per frame	$10ms$
processing time of video player per frame	$15ms$
network dispatching delay	$10ms$
number of packets per frame	65
number of database accesses to get a frame from the video server	12
number of users in the system	from 5 to 10

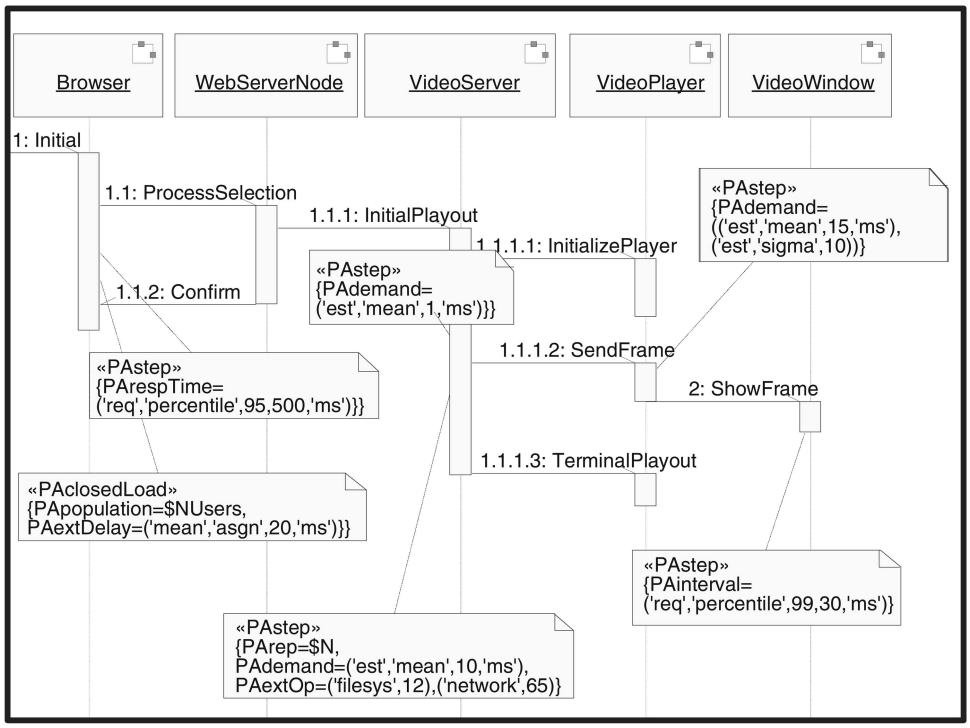

Fig. 12. Performance annotations example.

and validated against real values in working domains of system parameters. Then, the validated model can be used for performance prediction in domain ranges that cannot be currently inspected on the real system.

In the first case the estimation task is much easier, as it can rest on monitoring and extrapolation techniques [26]. In the second case, different techniques can be adopted for different categories of parameters, and we provide here some examples.

If the system is under development, it is reasonable that the target platform has not yet been determined. Moreover, performance analysis is often conducted before deployment to find the optimal configuration of the platform (e.g., number and characteristics of physical devices). Several alternative platforms can be available; thus, in these cases, the simulations of the same software side on different resource sides can be very useful for choosing among the alternative platforms. In addition, worst-case and best-case analysis can be conducted by devising, respectively, minimal and ideal platform architectures. This type of analysis provides a range of system performance and, if unsatisfactory, revision of the software architecture may be necessary.

The amount of resources needed to perform an action in the software side is often the hardest parameter to estimate when the system is under development. It depends on the granularity of the software model in that a step may correspond to a single instruction (if the software model is in an advanced development phase) or even to an entire service execution (if the model grain is still coarse). Note, however, that performance analysis before system deployment is often aimed at detecting critical component/subsystems (i.e., system bottlenecks) that need to be refined. Hence, absolute values of performance indices do not need to be tightly representative of the real system performance after deployment, whereas the ratios among performance values have to be maintained. For example, a performance expert does not expect that an analysis conducted on a system before deployment would produce the same response times that will be experienced on the system. The expert is interested in the trustability of the shape of a response time curve, for example, obtained from the analysis while varying the workload intensity. Suitable values of needed resources can be estimated, either based on the analyst's experience, or by extrapolated data from previous versions of the same software system, or by monitoring similar deployed systems.

The intensity of the workload is a typical input parameter in that only the range of values needs to be estimated. The model will be run under several sampled values inside the

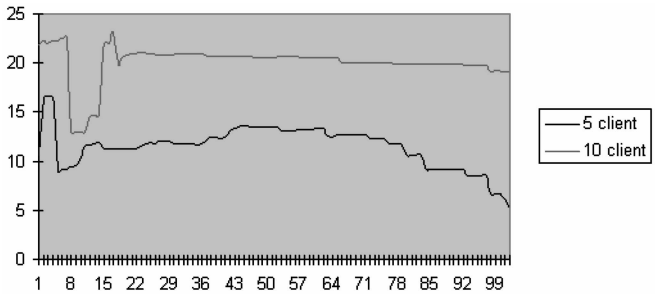

Fig. 13. Response time (in seconds) versus each resource request.

range. Ranges of workload can be estimated either by exposing early rapid prototypes of the system to users (this is the typical case of Web applications) or from experiences on similar systems developed previously.

3.3.2 Effort to Integrate Software and Platform Sides

The actions required to integrate software and resource sides, as described in the previous section, affect the transparency concept since they require a certain amount of effort and imply the modification of the software model. However, the proposed resource request standardization and the single processing node access point provided by the *Main Dispatcher* mitigate the global impact on software side. The "annotations" do not modify the basic structure and the behavior of software components. They can be "plugged-in" for performance analysis and then "unplugged" to come back to the original software model. In this sense, the technique we are describing is not free but, considering the importance of the performance analysis, it requires a relatively small and acceptable effort. Most importantly, the software designer can provide performance analysis using a well-known methodology and technology. From an implementation point of view, tool support can be helpful. Using RRT scripting support, we have implemented a user interface to fully assist designers in all steps. The designer indicates the point where the annotation must be inserted, selects the resources required, and indicates the estimated amount of such resources and the probes to be activated. Then, the script creates everything that is needed: the processing node (comprehensive of the required resources), the ports and the connectors between software components and processing node, and the instructions to send the resource request message. Thus, we compensate for the lack of transparency of the methodology by exploiting the tools capability.

3.4 Running Simulation and Analyzing Results

The running simulation step is strictly related to the simulation and model execution environment that the supporting tool provides. Assuming that several preconditions are satisfied (such as a well-defined target configuration and the correctness of model and included code syntax), in RRT this task is made extremely easy by simply issuing the "run" command . Actually, in RRT, it is possible to build and execute a model on different platforms (e.g., Windows with Visual C++ 6.0 or Linux with gcc). At the end of a simulation run, log files generated during the execution by the activated probes can be collected to evaluate the performance of the software model on the given platform model.

Part of the simulation results of the CSVSS example have been graphically represented in Fig. 13. The response time of the system has been reported vs. a certain number of user requests. The request number appears on the x-axis and the corresponding response time is reported on the y-axis under two different system workloads (i.e., 5 and 10 active users). The response time of each user request has been logged in the *Main Dispatcher* by carrying the difference between the time of request formulation and the time of request satisfaction. The expected increase of the response time under a heavier workload is easily visible in Fig. 13.

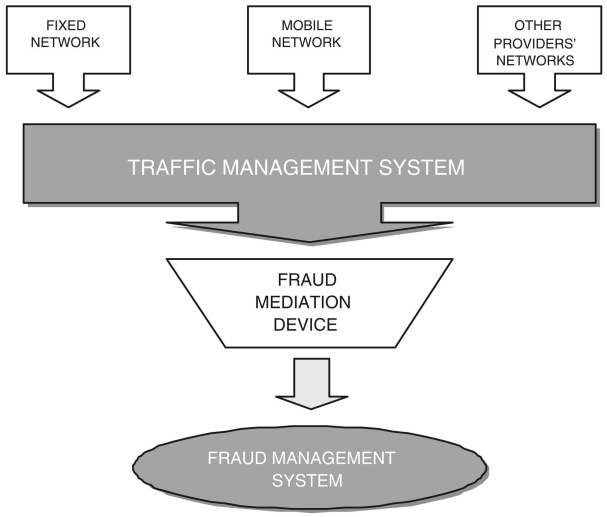

Fig. 14. The Fraud Mediation Device and its environment.

4 Validating the Framework against Real Case Studies

In this section, we show the application of our framework to two case studies from the real world, which are a Fraud Mediation Device and an SDH Telecommunication System. These are running systems, already implemented and deployed, that we have reverse-engineered to build the UML-RT models that we needed for performance analysis. Even though these are not thorough experiences on the whole software life cycle, they allowed us to validate the model construction and solution. For sake of readability, we propose a complete modeling and evaluation process for the first case study, whereas we only present numerical results obtained on the second one. Modeling aspects of the latter can be found in [8].

4.1 Fraud Mediation Device (FMD)

A Fraud Mediation Device (FMD) is a software system that filters communication data files collected from fixed and mobile devices. The main goal of a FMD is to preprocess and reorganize data so that they are ready to be processed from a Fraud Management System (FMS). In Fig. 14, an FMD is shown along with its connections to other subsystems which interact with it. The Traffic Management System (TMS) collects raw traffic data from the networks and dispatches data files to different subsystems (such as billing devices, fraud mediation devices, etc.). The FMD preprocesses data critical for fraud detection by following logical rules coming from a knowledge-based system (e.g., filtering, formatting, and integration rules). Preprocessed

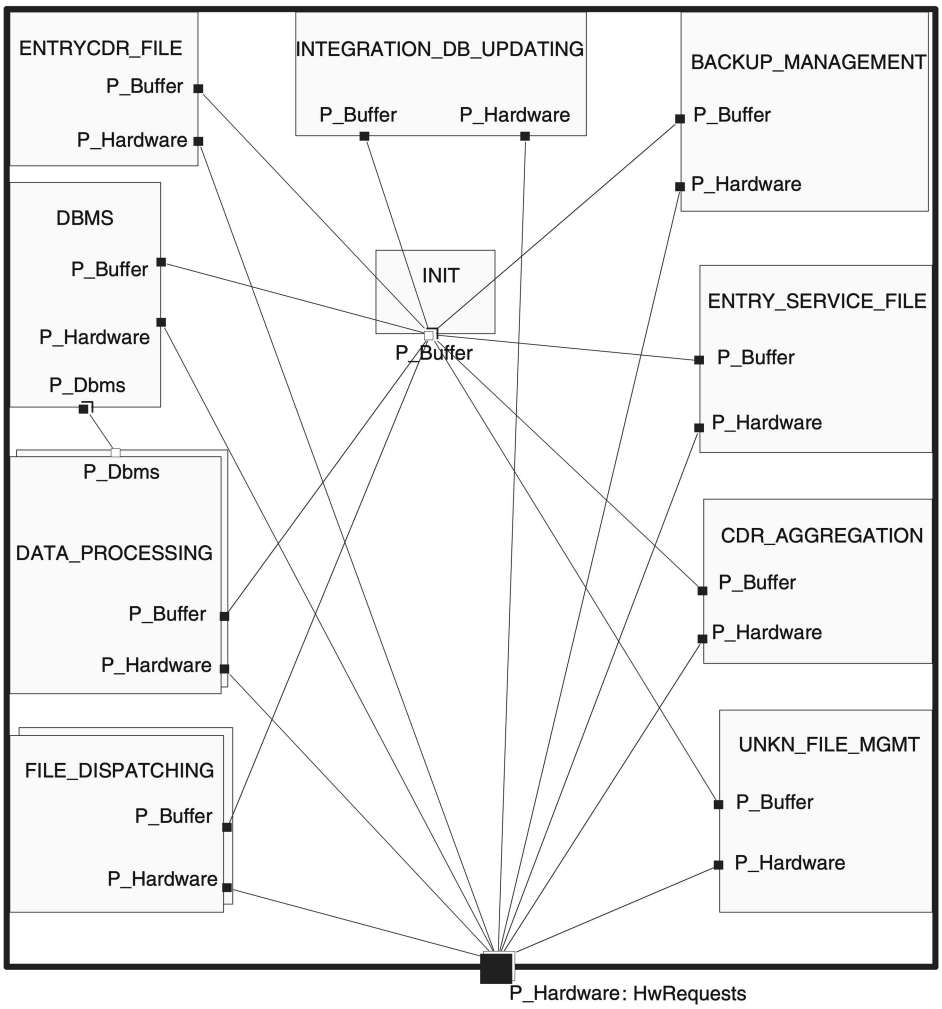

Fig. 15. The UML-RT Capsule Diagram of the FMD software side.

data is then provided to the FMS to perform the actual detection and recovery from fraud attempts.

4.1.1 FMD Software Side Modeling

We have modeled the FMD in UML-RT and, in Fig. 15, we show its Capsule Diagram.

We remark that Data Processing and File Dispatching capsules can have a multiplicity greater than one, because a pair of these component instances is generated for each type of traffic flow in the network.

Two additional capsules (that are not part of the FMD system) appear in Fig. 15: INIT manages simulation routines, such as the initial configuration setting (e.g., setting the number of instances of Data Processing capsule), and the capsule activation and synchronization. DBMS has been explicitly created to model the queries to the system DB.

4.1.2 FMD Platform Modeling

In order to apply our approach, the UML-RT software model has to be integrated with a platform model. The system that we have modeled was deployed on a HP AlphaServer GS1280 platform with the following configuration: SMP, cache-coherent Non-Uniform Memory Architecture (ccNUMA); 8 Alpha EV7 1,150 MHz processors; 32 GB RAM (4 GB/CPU); SUN-based distributed File System; Compaq Tru64 UNIX 5.1B Operating System; DBMS Oracle Server 9i Real Application Cluster.

As expected, it was necessary to extend the library of resource prototypes for this experiment, in particular for what concerns the memory management system. Based on the current platform architecture, in fact, memory size could be a crucial element in the performance analysis of such a system. Toward this goal, the SimpleRam and PagingRam resources have been created along with a Memory Internal Dispatcher [8].

4.1.3 Experimental Results

For the sake of readability, we do not report here details of the monitoring task on the deployed system. This task has allowed us to assign sizes of resource requests and intensity of the workload on the basis of the average amount of traffic per type of flow over the network. It has been observed that the traffic is heavily dependent on the weekday, so the results that we report refer to different days during one average traffic load week. The parameter that we consider here is the average utilization over the eight CPUs. In the figures that report the results of this experiment, we have placed, on the x-axis, the hours of a day, and, on the y-axis, the average utilization over the eight CPUs expressed as a percentage (i.e., utilization of 40 means that each CPU is utilized, on average, for 0.4 of its potential).

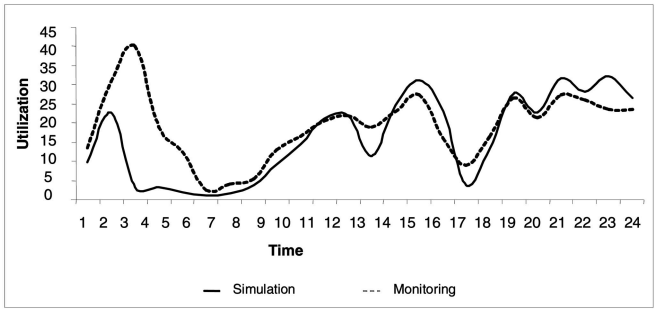

Fig. 16. CPU utilization on 5 November 2004.

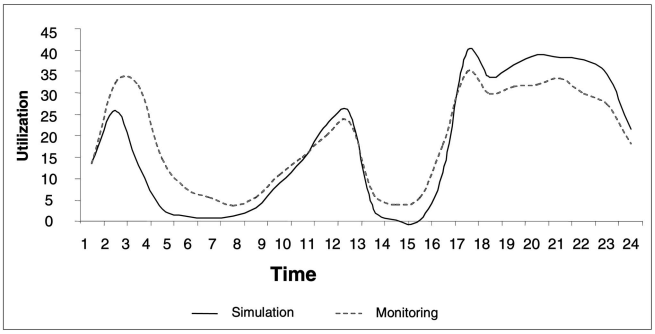

Fig. 17. CPU utilization on 26 October 2004.

The results of the first experiment in Fig. 16 show that the simulated model, after a startup interval (about six to seven hours of simulated time), where values do not nicely match with real system measures, is able to reproduce the monitored CPU utilization behavior of the real system over 24 hours in a day. In fact, the average error throughout the day is about 32 percent, whereas the average error after 9 a.m. drops to 19 percent (with a maximum error in this interval of 23 percent). However, beyond any numerical consideration, a relevant result is that the shapes of the curves are very similar.

The second experiment in Fig. 17 shows a better result of the simulated model; even in the startup interval the model values are not far from the real ones. Despite this better behavior in terms of shape, we have experienced still a difference in the numerical values. The difference holds around 38 percent for the whole day, whereas the average error after 8 a.m. drops to 29 (with a maximum error in this interval of 33 percent).

The differences experienced on numerical values originate from the intrinsic management of simulation time in UML-RT [8]. We have removed this problem by introducing a new time management technique that have been used in the simulation of the SDH system presented in next section.

The detection of a startup interval is based on empirical observation of performance results. We interpreted the larger error in the first hours as a startup time for the simulation, where random number generation was likely not yet in a steady state. The length of the startup interval obviously depends on the variance in the workload data. In fact, for such an FMD system, we have modeled in the workload the differences in the sources of data records (e.g., phone calls have a very different frequency with respect to short messages).

After model validation, we have used the model to predict the system performance under stress. We have simulated the model in the setting of the first experiment (i.e., the one in Fig. 16) while increasing the load in terms of traffic flow intensities. Our goal was to detect the saturation point for the CPU (i.e., utilization very close to 1). In Fig. 18, we report the results that we have obtained while multiplying the system load by 1.5 and by 3.

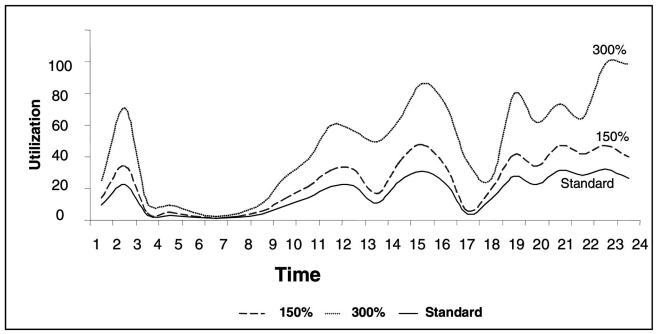

Fig. 18. CPU utilization on 5 November 2004 under stressing loads.

TABLE 2
Performance Indices

PING (pps)	25	50	75	100	105
MC TEST (%CPU)	11.56	22.68	33.72	43.86	46.18
Estimated (%CPU)	10.9	21.9	32.8	43.8	46
Δ %CPU	0.9	0.78	0.92	0.06	0.18

The utilization obviously increases, as expected, even though a 300 percent load does not saturate the system, except at the very end of the day.

4.2 SDH Telecommunication System (STS)

We have applied our approach to the Siemens SURPASS hiT 70xx product family [32]. The aim of this case study was to check the correctness of the simulation once solved the problems related to the RRT time management [8]. To perform such verification, we compare the performance results obtained by applying our performance analysis methodology and the ones measured during the system tests of an existing equipment. We have focused on one of the system verification tests executed to verify the equipment compliance to a nonfunctional requirement related to the amount of the CPU load induced by the traffic over the telecommunication management network (TMN). We have reported here a simplified model to obtain comparable results; however, an extensive description of this case study is available in [6], [8].

The simulation results have been compared with the values obtained by a real test. Some experimental results are shown in Table 2: The "Estimated" row shows the ones computed using the UML-RT simulation model, and the "MC TEST" row shows the mean values obtained over several test replications. The columns represent the different CPU loads evaluated for different numbers of ping packets per second (up to 105 pps that saturate each DCC link). The 100 pps value is the calibration point used to tune the model. The range considered for the pps value is realistically wide for actual systems. The length of the simulations has never exceeded a few minutes, and this is a promising result for the approach scalability. The difference between estimated and measured values is less than 1 percent on all the measurement points.[3]

3. The apparent underestimation of the values is, from our experience, due to characteristics of this specific example.

5 Lessons Learned and Discussion

In this section, we describe our experience with the approach on real case studies. The section is split in two parts that refer to different issues: First, we discuss the relationships of our approach with the software life cycle activities; then, we report our impression on the RRT simulation engine.

5.1 Our Approach within the Software Life Cycle

Automation in performance analysis is an essential factor to make this activity acceptable to software designers. The approach presented in this paper is aimed at automating the integration of software models with platform models and to provide feedback to software designers, without making them involved in the performance evaluation process.

In this direction, we find UML-RT a quite flexible notation to represent resource prototypes in a general way. Each prototype can be instantiated by simply setting a few parameters, such as a CPU speed in terms of MIPS. Moreover, the possibility of integrating the models within the same notation/tool guarantees automated consistency and synchronization between the software/platform model and the performance model.

Our approach is not tied to any phase of the software life cycle. Independently of the level of detail achieved, whatever (UML-RT) software model can be integrated with a platform model and its performance can be analyzed. Obviously, the accuracy of the results also depends on the level of knowledge of the system under development. As an example, we have experienced that the amount of resources needed for a certain element of the software model to accomplish its task is a quite critical parameter because it heavily affects the delay due to the resource utilization and the contention with other software elements. Therefore, while progressing in the development process, the software elements will be ever more fine grained (i.e. from components through objects to instruction) and the resource requests must be appropriately refined.

In Section 3, we have specifically discussed the effort needed to perform each step of the methodology. However, the whole effort also depends on the software process model adopted by the designer team. Once given an initial model, each refinement in the software model and/or in the target platform may induce more or less significant modifications on the remainder of the model. For example, if an evolutionary approach is adopted for software development, each iteration brings a certain number of changes in the software model and a reevaluation of the model to obtain the new performance indices. Therefore, we can characterize this type of effort with two parameters:

- The *total number of iterations* made on the software model, from its initial version to the last one. This quantity determines the number of times that the integrated model has to be simulated, with a consequent cost in terms of time.
- The *amount and type of changes* introduced in each iteration. From our experience, some kinds of changes do not induce heavy modifications in the platform model and in the code to require resources, whereas other kinds do. For example, if a software components is split into two subcomponents that are allocated on the same site as the original component, then the resource demands have to be updated with respect to the same (amount and type of) resources. Vice versa, if the allocation of the two subcomponents changes, then the resource demands have to be reestimated for different sets of available resources.

Finally, we report an observation from [13]: "An important requirement for the efficiency of the performance engineering process itself is that implementation decisions can be easily changed in the chosen formalism. This allows to quickly evaluate a set of different implementation designs and to select the best alternatives." Upon the application of our approach to real case studies, we believe that most of the effort is spent building an initial model. If the initial model is well-founded in all its main aspects (i.e., software modeling, platform modeling, resource demands), model changes can be easily implemented. In the RRT implementation that we have presented here, this is also due to the good support that the tool provides for model updates.

5.2 On the RRT Simulation Etnvironment

In this section, we briefly comment on our experience with the RRT simulation environment.

The main advantage of adopting RRT was the availability of an expressive and sound notation to model platform models and obtain multiple performance indices through probes (e.g., response time, throughput, software and hardware utilization, etc.). On the other hand, the main problem we faced with our initial approach to RRT was a lack of documentation (besides brief tutorials integrated in the tool) about the libraries of low-level functions that we have modified, for example, to introduce the management of the simulation time [8] and to handle the resource requests.

Summarizing below, we report the RRT characteristics that more heavily affected our implementation decisions:

- *Code on actions*: Based on the simple user interfaces that we have built, users are not aware of the code added on actions for resource requests; they only have to specify the *amounts* of resources that each critical action needs. With this solution, the only mistake that can be made is in the specification of these quantities.
- *Probe specification, results collection, simulation length*: The probes that we have introduced provide a significant support to easily specify the values to be collected and the locations where they have to be probed; in addition, RRT provides low level functions that can be exploited to this goal, thus leaving few limitations in the expressiveness of probes. On the other hand, confidence intervals are not provided among the RRT capabilities, they can be introduced into the probes, but we have not implemented this feature. Through our probes, we have monitored the evolution of simulation indices; thus, we have been able to set the simulation lengths so that a satisfactory accuracy of simulation results was achieved.
- *Workload specification*: There is no predefined utility to specify the simulation workload in RRT; as shown in Fig. 2, we have explicitly introduced in all our models a *Workload Generator* capsule that has a standard behavior specified by its statechart. Depending on the user specifications (given in terms of classes of jobs, open/close workload, interarrival/think time, etc.), jobs are generated in the software model; the workload on the platform model is

induced from the resource requests specified in the software side.
- *Simulation debugging*: A visual step-by-step execution mode can be set in the RRT environment; a set of selected statecharts can be shown on the screen and their evolution can be monitored while the simulation runs.

Finally, we would like to remark again that RRT is only one of the potential environments available for implementing our methodology (see, for example, HyPerformix [27]). Our choice mostly originated from the UML support that RRT offers, as UML today is a de facto standard for the software development activities. Therefore, the advantage that RRT offers is a wide integration in the real world software development processes, as experienced in our case studies.

6 Conclusions

The work presented in this paper is a long-time result of the study that we have conducted on the possibility of automating the integration of software models and platform models for performance analysis. In particular, we have been interested in studying this problem within the same notation. The rationale behind this goal is that building a performance model in the same notation used for software modeling may be more easily understood (and accepted) by the software developers compared to model transformation approaches. It was obvious from the very first moment that the price for achieving this goal is that the performance model cannot be analytically solved, and must be simulated.

We have implemented our methodology with the support of the Rose Real Time tool. In particular, we have exploited the characteristics of the UML-RT notation to represent platform characteristics and resource requests. These elements constitute the building blocks to transparently integrate software and platform models into a performance model.

The solution of implementation issues (such as the simulation time management), raised during the application of our approach to real case studies, has consolidated the framework that indeed has given satisfactory results, as illustrated in this paper.

Our methodology offers a sensitivity with respect to the software side and with respect to the platform side of the model, thus providing a wide variety of potential solutions to performance problems within the same notation and tool. A faster CPU can always be bought to temporarily solve a performance problem, but, in order to provide a more systematic solution to performance problems, we are more interested (as software engineers) in refining the software architecture before modifying the platform. To do that, it is crucial to have an integrated software/platform model.

A current limitation of this approach (on which we intend to work in future) resides in the limited ability to model complex middleware logic, which (at the moment) is all embedded into the two layers of dispatchers. We intend to introduce new prototypes that can be composed to model advanced middleware characteristics.

The prototype library is continuously growing, as every case study that we consider originates the need of modeling specific resources, besides the general ones. A smart approach to the resource modeling, in the future, would be to model a small set of resource prototypes, from which it is possible to derive other types of resources. In fact, a hierarchical structure of the prototype library would reflect the real world setting, and is an ambitious goal.

In addition, the implementation of confidence intervals to control the simulation runs is part of our short-term goals.

As a medium-term goal, we also intend to investigate the possibility of extending our approach to other simulation environment based on UML 2. In fact, all the features of UML-RT that are used in this research are now part of UML 2. At the moment, we are considering the porting of the approach to TAU Telelogic that supports UML 2 constructs and provides interesting model simulation facilities.

Finally, we keep searching in the real world for companies that intend to experiment our methodology from the beginning of the software lifecycle. This type of experiment would give us a clear quantification of all the efforts to be spent.

Acknowledgments

The authors would like to thank the reviewers for their comments that have allowed to consistently improve the paper scope, contents and presentation. The authors would also like to thank all the students that, in the past, have contributed to achieve these results. Finally, the authors thank Jessie Ostrowski for her accurate review of the paper text.

References

[1] H.H. Ammar, V. Cortellessa, and A. Ibrahim, "Modeling Resources in a UML-Based Simulative Eenvironment," *Proc. ACS/IEEE Int'l Conf. Computer Systems and Applications,* 2001.
[2] L.B. Arief and N.A. Speir, "A UML Tool for an Automatic Generation of Simulation Programs," *Proc. Second Int'l Workshop Software and Performance,* 2000.
[3] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Trans. Software Eng.,* vol. 30, no. 5, pp. 295-310, May 2004.
[4] S. Balsamo and M. Marzolla, "A Simulation-Based Approach to Software Performance Modeling," *Proc. European Software Eng. Conf./ACM SIGSOFT Symp. Foundations of Software Eng.,* 2003.
[5] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models," *Proc. Third Int'l Workshop Software and Performance,* 2002.
[6] V. Cortellessa and M. Gentile, "Performance Modeling and Validation of a Software System in a RT-UML-Based Simulative Environment," *Proc. Int'l Symp. Object-Oriented Real-Time Distributed Computing,* 2004.
[7] V. Cortellessa, P. Pierini, and D. Rossi, "On the Adequacy of UML-RT for Performance Validation of an SDH Telecommunication System," *Proc. Int'l Symp. Object-Oriented Real-Time Distributed Computing,* 2005.
[8] V. Cortellessa, P. Pierini, and D. Rossi, "Software Performance Validation in UML-RT," Technical Report TRCS 002-2007, Dipartimento di Informatica, University of L'Aquila, www.di.univaq.it/cortelle/docs/UMLRTreport. pdf, 2007.
[9] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison Wesley, 2000.
[10] H. Harreld, "NASA Delays Satellite Launch after Finding Bugs in Software Program," *Federal Computer Week,* www.fcw.com, 1998.
[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] *Model-Driven Architecture Guide,* omg/2003-06-01, J. Miller, ed., www.omg.org/mda, 2003.
[13] A. Mitschele-Thiel and B. Muller-Clostermann, "Performance Engineering of SDL/MSC Systems," *Computer Networks,* vol. 31, pp. 1801-1815, 1999.
[14] I. Ober, S. Graf, and I. Ober, "Validating Timed UML Models by Simulation and Verification," *Proc. Int'l Workshop Specification and Validation of UML Models for Real Time and Embedded Systems (UML '03 satellite event),* 2003.
[15] R. Pooley and C. Kabajunga, "Simulation of UML Sequence Diagrams," *Proc. 14th UK Performance Eng. Workshop,* 1998.
[16] J.A. Rolia, K.C. Sevcik, "The Method of Layers," *IEEE Trans. Software Eng.,* vol. 21, no. 8, pp. 689-700, 1995.
[17] H. Ryan, "Too Big to Fail?" *Outlook 2000,* no. 1, pp. 3-9, 2000.
[18] P.P. Sancho, C. Juiz, and R. Pugjaner, "Integrating System Performance Engineering into MASCOT Methodology through Discrete-Event Simulation," *Lecture Notes in Computer Sciences,* vol. 3236, pp. 278-292, 2004.
[19] B. Selic, "Some Unresolved Problems in Real-Time Design," *Proc. Int'l Symp. Object-Oriented Real-time Distributed Computing,* 2004.
[20] B. Selic, "On Software Platforms, Their Modeling with UML2, and Platform-Independent Design," *Proc. Int'l Symp. Object-Oriented Real-time Distributed Computing,* 2005.
[21] B. Selic, "Using UML for Modeling Complex Real-Time Systems," *Lecture Notes in Computer Sciences,* vol. 1474, pp. 250-260, 1998.
[22] M. Woodside, C. Hrischuk, B. Selic, and S. Bayarov, "A Wideband Approach to Integrating Performance Prediction into a Software Design Environment," *Proc. First Int'l Workshop Software and Performance,* 1998.
[23] M. Woodside, C. Hrischuk, B. Selic, and S. Bayarov, "Automated Performance Modeling of Software Generated by a Design Environment," vol. 45, *Performance Evaluation,* pp. 107-123, 2001.
[24] M. Woodside, D.C. Petriu, D.B. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by Unified Model Analysis (PUMA)," *Proc. Fifth Int'l Workshop Software and Performance,* 2005.
[25] S. Yacoub, A. Ibrahim, H.H. Ammar, and K. Lateef, "Verification of UML Dynamic Specification Using Simulation-Based Timing Analysis," *Proc. Sixth Int'l Conf. Reliability and Quality in Design,* 2000.
[26] C. Yilmaz, A.S. Krishna, A. Memon, A. Porter, D.C. Schmidt, A. Gokhale, and B. Natarajan, "Empirical Software Engineering: Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems," *Proc. Int'l Conf. Software Eng.,* 2005.
[27] "Engineering Performance Early in the Application Life Cycle," white paper, HyPerformix, www.hyperformix.com/2006-aps-proceedings/Documents/Designer-white-paper.pdf, 2005.
[28] *Proc. First Int'l Workshop Software and Performance,* portal.acm.org, 1998.
[29] "UML Profile for Schedulability, Performance, and Time Specification," formal/2005-01-02, OMG full specification, http://www.omg.org/technology/documents/formal/schedulability.htm, 2007.
[30] IBM Rational Rose Real Time, www.rational.com, 2007.
[31] Real-Time and Distributed Systems Group, Carleton Univ., www.sce.carleton.ca/rads/rads.html, 2007.
[32] Surpass Series Products, Siemens Information and Comm. Network, www.siemens.com/surpass, 2007.

Vittorio Cortellessa is an associate professor in the Computer Science Department at Università dell'Aquila, Italy. Prior to joining Università dell'Aquila, he was a postdoctoral fellow with the European Space Agency, Rome; a research associate in the Computer Science Department at Università di Roma "Tor Vergata;" and a research assistant professor in the Lane Department of Computer Science and Electrical Engineering at West Virginia University, Morgantown. He has been involved in several research projects in the areas of performance and reliability analysis of software/hardware systems, model-driven engineering, nonfunctional software validation, and software specification of fault-tolerant systems, which are his current main research interests. He has published about 50 journal and conference articles on these topics. He has served and is currently serving in the program committees of conferences in his research areas.

Pierluigi Pierini is an electronic engineer working at Technolabs SpA R&D Laboratory, Italy, in the field of telecommunication systems. He previously worked for companies such as Selenia SpA in the ATC field and Italtel SpA and Siemens AG in the TLC field. His research interests include system architecture of embedded systems and several aspects of software engineering, such as requirements analysis, model-driven engineering, test design and performance analysis. He is the Technolabs associate responsible for the research programs on some of these topics developed with the ISTI of Pisa and, still in progress, with the Computer Science Department of the Università dell'Aquila.

Daniele Rossi received the MSc degree in computer science from the Università dell'Aquila, Italy, in 2004. He was a postdegree fellow at Technolabs SpA, where he worked on UML software modeling and performance analysis of telecommunication systems. Currently, he is a software consultant for several software and service companies in Rome. His research interests include software modeling and performance analysis of software/hardware systems.