To appear in proceedings of the Fourth International Conference on the Quality of Software Architectures (QoSA 2008). Germany, October 14-17, 2008

Model-Driven Performance Analysis

Gabriel A. Moreno and Paulo Merson

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{gmoreno,pfm}@sei.cmu.edu

Abstract. Model-Driven Engineering (MDE) is an approach to develop software systems by creating models and applying automated transformations to them to ultimately generate the implementation for a target platform. Although the main focus of MDE is on the generation of code, it is also necessary to support the analysis of the designs with respect to quality attributes such as performance. To complement the model-to-implementation path of MDE approaches, an MDE tool infrastructure should provide what we call model-driven analysis. This paper describes an approach to model-driven analysis based on reasoning frameworks. In particular, it describes a performance reasoning framework that can transform a design into a model suitable for analysis of real-time performance properties with different evaluation procedures including rate monotonic analysis and simulation. The concepts presented in this paper have been implemented in the PACC Starter Kit, a development environment that supports code generation and analysis from the same models.

1 Introduction

Model-Driven Engineering (MDE) is an approach to create software systems that involves creating models and applying automated transformations to them. The models are expressed in modeling languages (e.g., UML) that describe the structure and behavior of the system. MDE tools successively apply pre-defined transformations to the input model created by the developer and ultimately generate as output the source code for the application. MDE tools typically impose domain-specific constraints and generate output that maps onto specific middleware platforms and frameworks [1]. MDE is often indistinctively associated to OMG's Model-Driven Architecture and Model-Driven Development.

The ability to create a software design and apply automated transformations to generate the implementation helps to avoid the complexity of today's implementation platforms, component technologies and frameworks. Many MDE solutions focus on the generation of code that partially or entirely implements the functional requirements. However, these solutions often overlook runtime quality attribute requirements, such as performance or reliability. Fixing quality attribute problems once the implementation is in place has a high cost and often requires structural changes and refactoring. Avoiding these problems is the main

motivation to perform analysis early in the design process. To complement the model-to-implementation path of MDE approaches, an MDE tool infrastructure should provide what we call model-driven analysis. The model to code path and the model-driven analysis path are notionally represented in Figure 1. The goal of model-driven analysis is to verify the ability of the input design model to meet quality requirements.

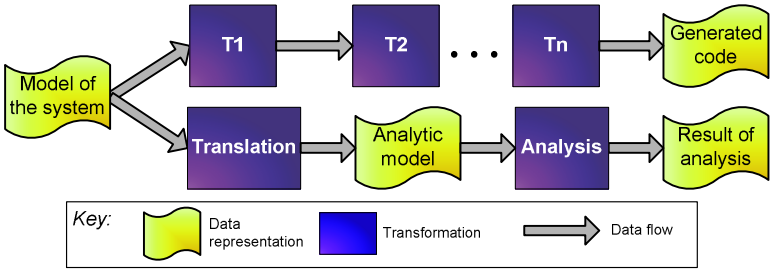

Fig. 1. Model-Driven Engineering and Model-Driven Analysis

The Software Engineering Institute has developed an MDE tool infrastructure that offers code generation along with various analytic capabilities. This tool infrastructure is called the PACC Starter Kit (PSK) [2]. It uses the concept of reasoning frameworks [3] to implement analytic capabilities. Several reasoning frameworks have been developed and applied to industry problems. This paper focuses on our performance reasoning framework, which analyzes timing properties of component-based real-time systems. The paper describes the model-driven approach used in the implementation of the reasoning framework.

The remainder of the paper is organized as follows. Section 2 introduces the concept of a reasoning framework and then describes the elements of our performance reasoning framework. Section 3 describes the intermediate constructive model, which is the first model created when analyzing an input design. Section 4 explains the performance model and how it is generated from the intermediate constructive model through a transformation called interpretation. Section 5 briefly describes how the performance model is used by different evaluation procedures to generate performance predictions. Section 6 shows an example of the application of the concepts described in the paper. Section 7 discusses related work and Section 8 has our concluding remarks.

2 Performance Reasoning Framework

A reasoning framework provides the ability to reason about a specific quality attribute property of a system's design [3]. Figure 2 shows the basic elements of a reasoning framework. The input is an architecture description of the system, which consists of structural and behavior information expressed in a modeling

language or any formally defined design language. Reasoning frameworks cannot analyze any arbitrary design. Furthermore, there is a tradeoff between the analytic power of a reasoning framework and the space of designs it can analyze. Reasoning frameworks restrict that space by imposing analytic constraints on the input architecture description. The analytic constraints restrict the design in different ways (e.g., topological constraints) and also specify what properties of elements and relations are required in the design.

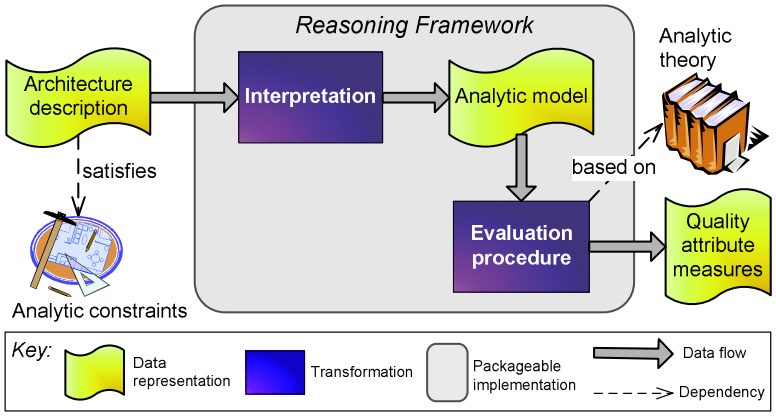

Fig. 2. Basic elements of a generic reasoning framework

The architecture description is submitted to a transformation called interpretation. If the architecture description is well-formed with respect to the constraints and hence analyzable, the interpretation generates an analytic model representation. This model is an abstraction of the system capturing only the information relevant to the analysis being performed. The types of elements, relations and properties found in an analytic model are specific to each reasoning framework. The analytic model is the input to an evaluation procedure, which is a computable algorithm implemented based on a sound analytic theory. The implementation of the evaluation procedure may be purely analytic, simulation-based or a combination of both.

Figure 3 shows the basic elements of our performance reasoning framework. Comparing to Figure 2, we see an additional step that translates the architecture description to a data representation called ICM. ICM stands for intermediate constructive model and it is a simplified version of the system's original design. The ICM is described in Section 3. The analytic model seen in Figure 2 corresponds to the performance model in Figure 3, which will be described in Section 4. The evaluation procedure box in Figure 2 corresponds to the performance analysis box in Figure 3, which is carried out by analytic and simulation-based predictors created based on rate monotonic analysis (RMA) [4] and queuing theory [5]. As described in Section 5, the predictors can estimate average and

worst-case latency of concurrent tasks in a system that runs on a fixed priority scheduling environment.

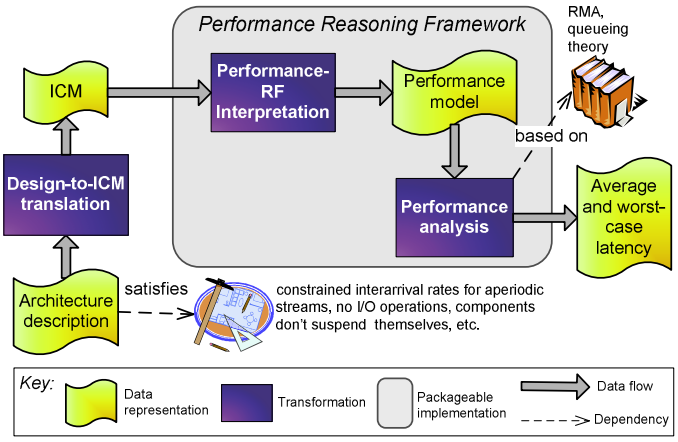

Fig. 3. Performance Reasoning Framework

The performance reasoning framework is packaged and independently deployed as an Eclipse plug-in [6]. Thus any Eclipsed-based tool used to model software systems in a parseable design notation could benefit from the performance reasoning framework by exporting their designs to ICM. Examples of such tools include: IBM Rational Software Architect, OSATE tool for the AADL language, Eclipse Model Development Tools (MDT) UML2, and AcmeStudio. When adding the performance reasoning framework plug-in to a modeling tool, the only thing one has to do is to create a class that implements the following method:

```
AssemblyInstance translateDesignToIcm(IFile designFile);
```

As expected, the implementation of this method is entirely specific to the design language representation used by the tool. We have implemented this method for the CCL design language [7] that is used in the PSK. An explanation of CCL or how to translate it to ICM is beyond the scope of this paper. But we should note that the design language used as input for the performance reasoning framework shall support the following elements and relations:

- components with input and output ports[1]. If the design language were UML, for example, UML components with required and provided interfaces could be used.
- special components called environment services (or simply services) that represent external elements of the runtime environment that the system interacts with. Clocks, keyboards, consoles, and network interfaces are examples

[1] In CCL, an input port is called "sink pin" and an output port is called "source pin"—these are the terms used in this paper.

of services. Like regular components, services may have sink and source pins. In UML, they could be represented as stereotyped UML components.
- a way to wire the components together, that is, to connect a source pin to a sink pin. In UML, a simple UML assembly connector could be used.
- a way to differentiate between synchronous and asynchronous interactions, as well as threaded and unthreaded interactions. In UML, stereotypes could be used to indicate these characteristics.
- a way to annotate any element with specific pairs of key and value. These annotations are used for properties required by the reasoning framework (e.g., the performance reasoning framework requires the priority of each component to be specified). In UML, annotations could be done with tagged values.

3 Intermediate Constructive Model

The architecture description that is the input to a reasoning framework (see Figure 2) is itself a constructive model of the system. However, it often contains many details that are not used in the performance analysis. For example, the state machine of a component may be used for code generation but is not needed by the performance reasoning framework. To remove details specific to the input design language and hence simplify the interpretation translation, we created the intermediate constructive model (ICM). The design-to-ICM translation (see Figure 3) abstracts the elements of the architecture description that are relevant to performance analysis to create the ICM.

Figure 4 shows the ICM metamodel. A complete description of the elements, properties and associations in the ICM metamodel is beyond the objectives of this paper. Here, we present the information pieces that are key to the performance analysis discussion in subsequent sections. The entry point to navigate an ICM is *AssemblyInstance*, which is the return type of the `translateDesignToIcm` method mentioned earlier. The system being analyzed is an assembly of components. In the ICM metamodel, a component is generically called *ElementInstance*. The use of the suffix "instance" avoids confusion when the input design language allows the definition of *types* of components. For example, in CCL one can define a component type called *AxisController*. Then a given assembly may contain two components (e.g., *axisX* and *axisY*) that are instances of that component type. The ICM for that assembly would show two *ElementInstance* objects with the same type (*AxisController*) but different names.

An *ElementInstance* object (i.e., a component) has zero or more pins (*PinInstance* objects in the metamodel). Each pin is either a *SinkPinInstance* or a *SourcePinInstance* object. A sink pin is an input port and when it is activated it performs some computation. The performance analysis is oblivious to most details of that computation. However, it is necessary to know what source pins (output ports) are triggered during the computation and in what order—that is given by the *reactSources* association between *SinkPinInstance* and *SourcePinInstance*. It is also necessary to specify the priority that the sink pin computation will have at runtime. Another important property of a sink pin is the execution

time for the corresponding computation, which is shown in the ICM metamodel as the *execTimeDistribution* association between *SinkPinInstance* and *Distribution*. A sink pin can be synchronous or asynchronous. If it is synchronous, it may allow concurrent invocations (reentrant code) or enforce mutual exclusion on the sink pin's computation.

Real-time systems, especially ones with aperiodic threads, sometimes exhibit different behavior depending on certain conditions of the environment or different pre-set configurations. For example, a system can be operating with limited capacity due to a failure condition, or a system can have optional "pluggable" components. These kinds of variability are represented in the ICM as *Scenario* objects. A scenario can be represented in the architecture description as annotations to the assembly and the sink pins that are active under that scenario.

Once the ICM for a given architecture description is created, the performance reasoning framework can perform the interpretation translation to generate the performance model, which is used as input to the performance analysis. These steps are described in the following sections.

4 Performance Model Generation

An important component of a reasoning framework is the interpretation process that transforms an architecture description into an analytic model. Section 2 showed that the architecture description is translated to an intermediate representation (ICM) in our performance reasoning framework. In this reasoning framework, interpretation starts with an ICM model and produces a performance model that can then be analyzed by different evaluation procedures.

4.1 Performance Metamodel

The performance metamodel (Figure 5) is based on the method for analyzing the schedulability of tasks with varying priority developed by Gonzalez Harbour et al. [8]. In this method, a task is a unit of concurrency such as a process or a thread. Tasks are decomposed into a sequence of serially executed subtasks, each of which has a specific priority level and execution time.

The root element of a performance model in our reasoning framework is *PerformanceModel*, which contains one or more *Tasks*. Unlike in the method by Gonzalez Habour et al., where all tasks are periodic, in our metamodel a task is either a *PeriodicTask* or an *AperiodicTask*. Periodic tasks are characterized by a period and an offset that is used to model different task phasings at startup. Aperiodic tasks, on the other hand, are tasks that respond to events that do not have a periodic nature. For that reason, they have an *interarrivalDistribution* to describe the event arrival distribution. For example, the events may follow an exponential distribution where the mean interarrival interval is 10ms. The *SSTask* models aperiodic tasks scheduled using the sporadic server algorithm [9], which allows scheduling aperiodic tasks at a given priority while limiting their impact on the schedulability of other tasks.

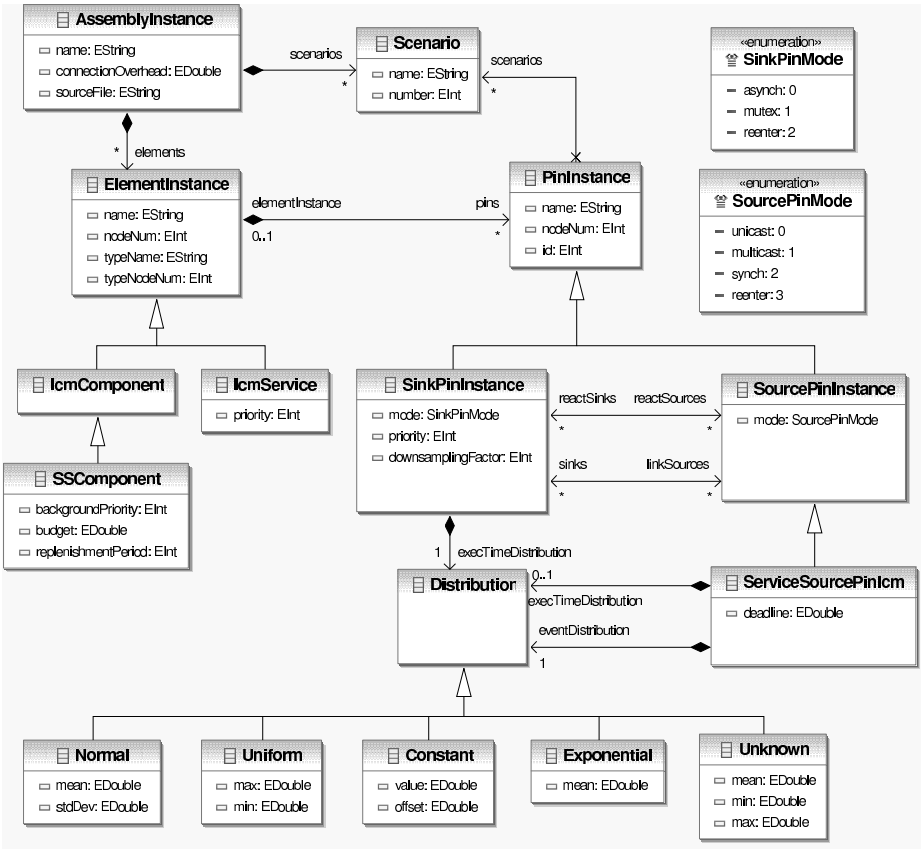

Fig. 4. ICM metamodel (notation: UML)

Tasks do not have an explicit execution time attribute because their computation is carried out by the subtasks they contain. Thus, each *Subtask* has an execution time and a priority level. The metamodel supports both constant and random execution times by providing different kinds of *Distribution*. The task is the unit of concurrency. That is, tasks execute in parallel—subject to a fixed priority scheduling policy—and within a task there is no concurrency.

4.2 From ICM to Performance Model

As depicted in Figure 2, the interpretation that generates the analysis model can only be carried out if the analytic constraints of the reasoning framework are satisfied by the design. The assumptions and analytic constraints of the performance reasoning framework are the following.

1. The application being analyzed executes in a single processing unit (i.e., in one single-core CPU or in one core of a multi-core CPU).

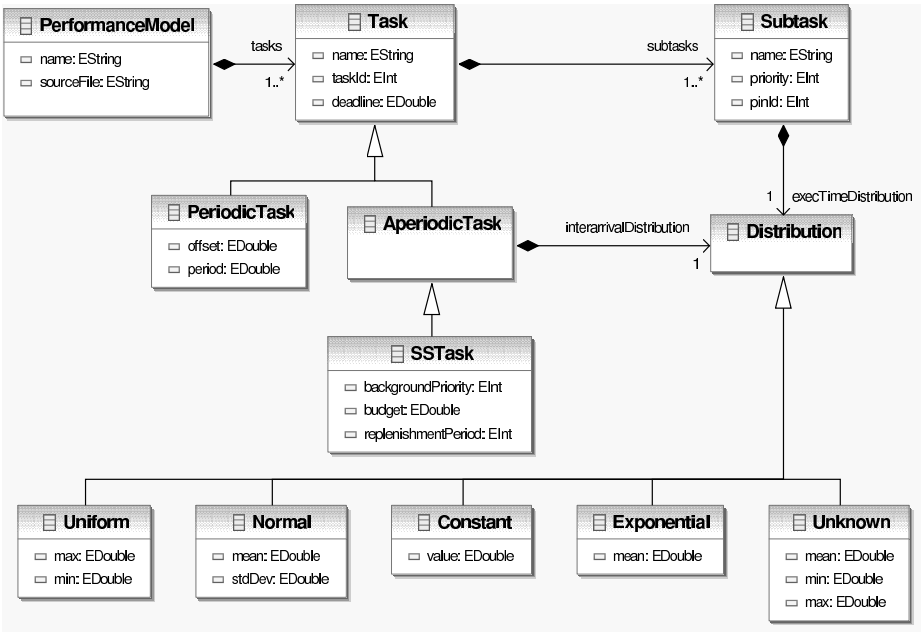

Fig. 5. Performance metamodel (notation: UML)

2. The runtime environment uses fixed-priority preemptive scheduling.
3. Components perform their computation first and then interact with other components.
4. Each sink pin in a component reacts with all the source pins in the reaction.
5. No two subtasks (or equivalently, sink pins) within a response can be ready to execute at the same time with the same priority level.
6. Priority of mutex sink pins is assigned according to the highest locker protocol.
7. Components do not suspend themselves during their execution.

Even though some of these constraints may seem too restrictive, they are the result of a process called co-refinement [10], a process that evaluates the tradeoffs between constraints imposed on the developers, the cost of applying the technology, and the accuracy of the resulting predictions. For example, constraint 3 makes the interpretation simpler because it does not require looking into the state diagram of the component, and also makes the use of the technology simpler because it requires fewer annotations to be provided by the developer.

The ICM and the performance model are different in several aspects. For example, the ICM can model a complicated network of computational elements, while the performance model only supports seemingly isolated sequences of them. The rest of this section describes the concepts guiding the transformation from ICM to performance model.

An event is an occurrence the system has to respond to. The tick of an internal clock and the arrival of a data packet are examples of events. A response is the computational work that must be carried out upon the arrival of an event [4]. The main goal of the performance reasoning framework is to predict the latency of the response to an event, taking into account the preemption and blocking effects of other tasks. In an ICM, a source of events is represented as a source pin in a service (i.e., a *ServiceSourcePinIcm*). Therefore, the goal translates into predicting the latency of all the components that are connected directly or indirectly to that service source pin. Since the response to an event is modeled as a task in the performance model, it follows that for each *ServiceSourcePinIcm* in the ICM, a *Task* needs to be created. Depending on the event interarrival distribution of the service source pin, the task will be a *PeriodicTask* or an *AperiodicTask*.

The next step, and the most complex one, is transforming a possibly multithreaded response involving several components into a sequence of serially executed subtasks. This transformation deals with two main issues: the internal concurrency within a response, and the blocking effects between responses.

Concurrency within a Response. Figure 6 shows an example of a response with concurrency. Component A asynchronously activates components B and C. Since B and C can execute concurrently, it seems they cannot be serialized as a sequence of subtasks. However, because they have different priorities, they will actually execute serially,[2] first the high priority component C and then the low priority component B, even when they are ready to execute at the same time. Thus, it is possible to determine the sequence of subtasks that represents the actual execution pattern. In order to do that, the design has to satisfy one analytic constraint: *each threaded component has to be assigned a unique priority within the response*. This is a sufficient but not necessary constraint because two components can have the same priority as long as they are never ready to execute at the same time. The interpretation algorithm flags situations where priority level sharing is not allowed.

Blocking between Responses. In Figure 7 there are two responses that use a shared component. Since this component is not reentrant, the responses can potentially block each other. This blocking is addressed in the performance model by using the highest locker protocol [4]. The shared component is assigned a priority higher than the priority of all the components using it. In addition, the component must not suspend itself during its execution. The benefit of these analytic constraints is twofold. First, they make the behavior more predictable because calling components are blocked at most once and priority inversion is bounded. Second, the non-reentrant component can be modeled as a subtask in each of the responses. There is no need to have special synchronization elements

[2] The analysis assumes the application runs in one processing unit (constraint 1). Therefore, threads that are logically concurrent are executed serially by the processor.

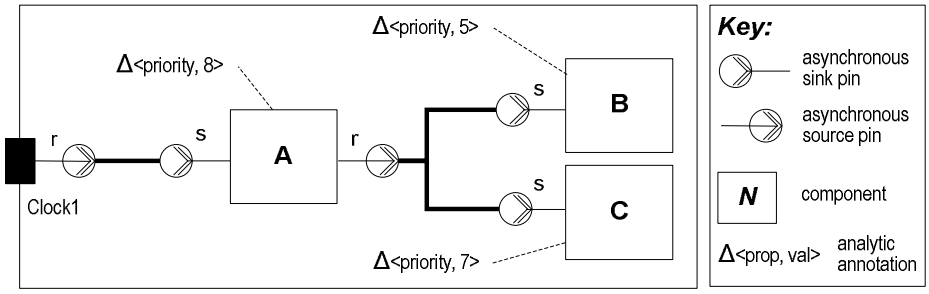

Fig. 6. Response with concurrency

in the model because the highest locker protocol and the fixed-priority scheduling provide the necessary synchronization.

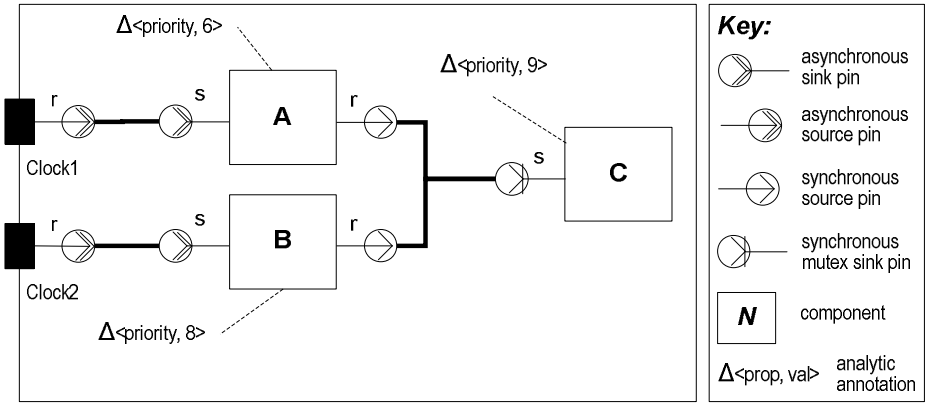

Fig. 7. Blocking between responses

Interpretation is a model transformation that translates from the ICM metamodel to the performance metamodel. This transformation has been implemented both using a direct-manipulation approach with Java and the Eclipse Modeling Framework (EMF), as well as with the ATL model transformation language [11]. Details of both implementations are beyond the scope of this paper.

5 Performance Analysis

The performance model produced by interpretation can be analyzed with a variety of evaluation procedures ranging from sound performance theories, such as RMA to efficient discrete-event simulators. The reason for this flexibility is

twofold. First, the evaluation procedures need neither be able to handle asynchronous calls nor keep track of call stacks because interpretation translates responses involving both into a sequence of serially executed subtasks. Second, evaluation procedures do not need an explicit notion of synchronization between tasks other than that resulting by virtue of the fixed priority scheduling.

Depending on its characteristics, a given performance model can be analyzed by some evaluation procedures and not by others. For example, models with unbounded execution or interarrival time distributions can be handled by a simulation-based evaluation, but not by worst-case analysis like RMA. For this reason, different evaluation procedures may dictate adhering to additional analytic constraints. In most cases, this is not a limiting constraint, but rather an enabler for predictability. For instance, when developing a system with hard real-time requirements where RMA will be used to predict worst-case response time, one should avoid designing responses with unbounded execution time.

The rest of this section describes the different evaluation procedures used in our performance reasoning framework. Some of them are third-party tools and have their own input format. In such cases, the performance model is translated to the particular format before it is fed to the tool for evaluation.

5.1 Worst-case Analysis

Worst-case analysis predicts the worst-case response time to an event in the system by considering the maximum execution time of the components and the worst preemption and blocking effects. Worst-case analysis in the performance reasoning framework is carried out with MAST, a modeling and analysis suite for real-time applications [12]. Among other techniques, MAST implements RMA with varying priorities [8].

5.2 Average-case Analysis

Average-case analysis computes the average response time to an event. Although this is achieved by discrete-event simulation in most cases, the performance reasoning framework also includes an analytic average-case analysis.

Simulation-based average-case analyses simulate the execution of a system taking into account the statistical distribution of interarrival and execution times. By collecting the simulated response time to thousands of arrivals, they can compute the average response time. In addition, they keep track of the best and worst cases observed during the simulation.

The performance reasoning framework supports three different simulation-based average case latency predictions. One of them is with Extend, a commercial general-purpose simulation tool that supports discrete-event simulation [13]. The simulation model is built out of custom blocks for Extend that represent the concepts in the performance model (i.e., tasks, subtasks, etc.). The Extend-based simulation can model several interarrival distributions and is able to simulate sporadic server tasks. Another simulation analysis is based on a discrete-event simulator called *qsim*. Being a special-purpose simulator, *qsim* is very efficient

and is able to run much longer simulations in less time. The last of the simulation-based analyses uses Sim_MAST, a simulator that is part of the MAST suite. Sim_MAST can simulate a performance model providing statistical information about the response time to different events.

The analytic average case analysis in the performance reasoning framework uses queuing and renewal theory to compute the average latency of a task scheduled by a sporadic server [5]. Although this method requires specific constraints such as exponential interarrival distribution, it provides an envelope for the average latency without the need to run long simulations.

6 Example

Figure 8 shows a screenshot of the PSK showing the CCL design diagram for a simple robot controller. The controller has two main components that execute periodically. The trajectory planner executes every 450ms and takes work orders for the robot. Using information from the position monitor, it plans a trajectory, translating the orders into subwork orders that are put into the repository. The movement planner, on a 150ms period, takes orders from the repository and converts them into movement commands for the two axes. The responses to these two periodic events have hard deadlines at the end of their period.

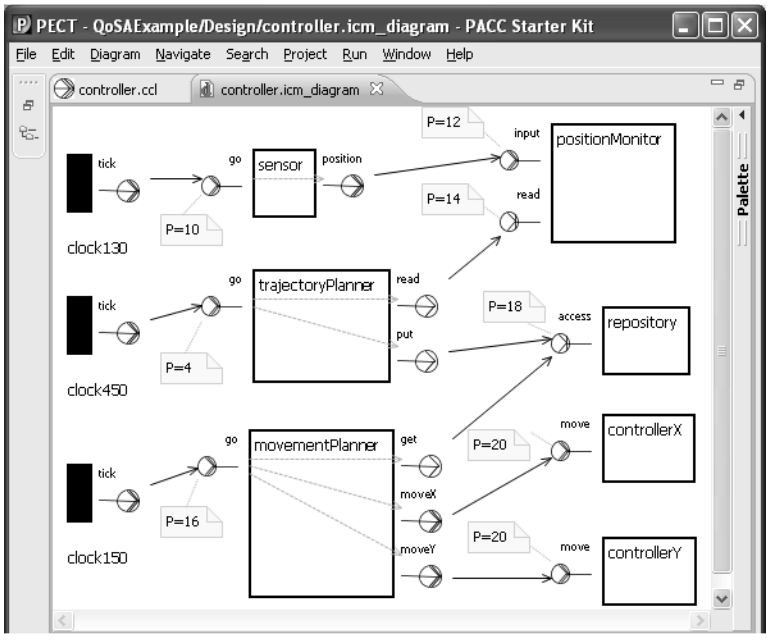

Fig. 8. Component and connector diagram for controller

When the performance reasoning framework is called, the user selects the desired evaluation procedure and enters some analysis parameters. After that, the design model is transformed to a performance model, which is then evaluated. Figure 9 shows the performance model created from the design of the robot controller. As previously described, the performance model does not have concurrency within the responses to the different events, and it does not involve explicit synchronization between the responses. Figure 10 shows the results of a worst-case latency analysis using MAST. In this particular case, the result viewer indicates that the response to the 450ms clock is not schedulable because it has a worst-case execution time that overruns its period.

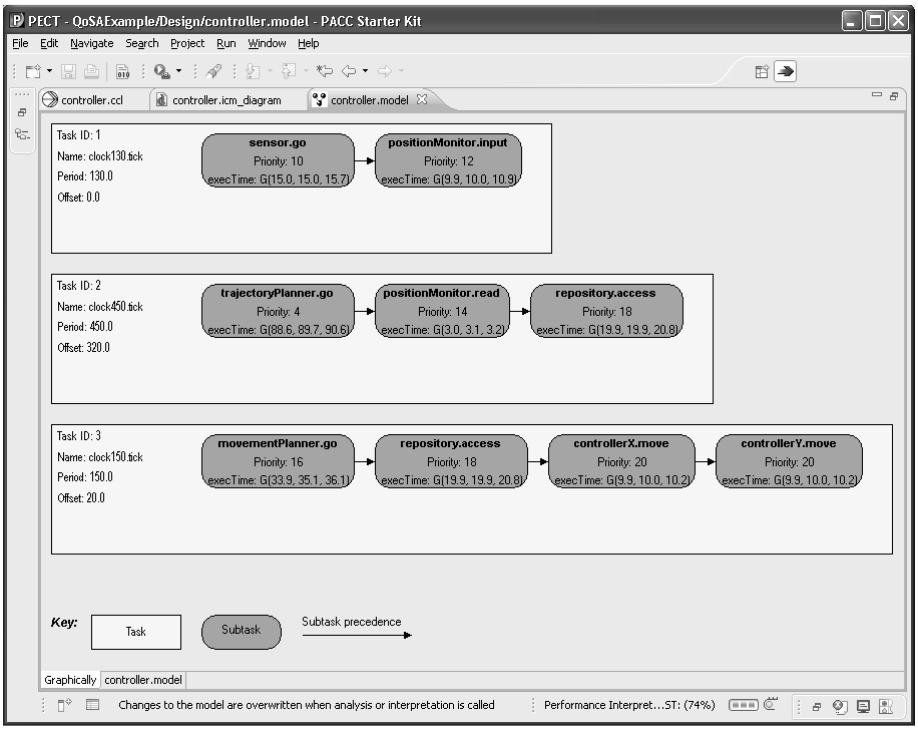

Fig. 9. Generated performance model for controller

7 Related Work

There has been recent work in integrating performance analysis into model-driven development approaches [14–16]. Here we describe the similarities and differences with some of them. Woodside et al. [17] and Grassi et al. [14] have proposed intermediate models—CSM and KLAPER respectively—to reduce the

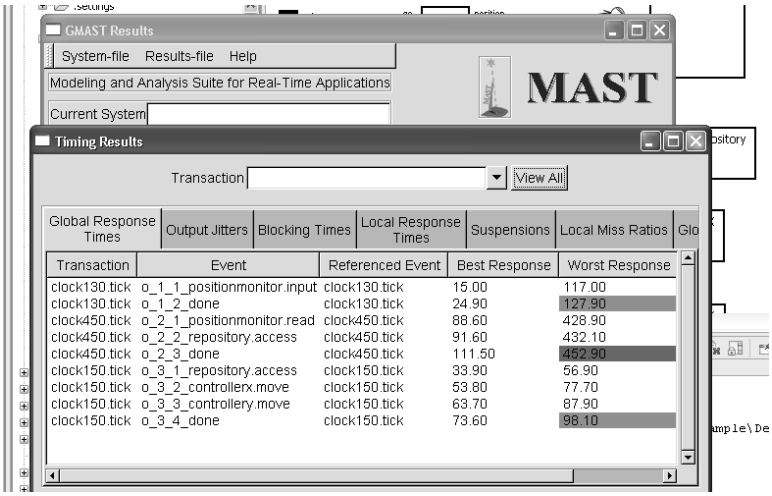

Fig. 10. Analysis results for the controller

semantic gap between the design models and the analysis models and enable the use of analysis tools with different design languages. In that regard, ICM has the same intent. However, ICM is only one element in our reasoning framework, serving as an input meta-model for one of the key elements in the reasoning framework, namely, the interpretation that transforms the design into a performance model. Since our approach uses two meta-models, they are respectively closer to the start and end of the model-driven analysis process. For instance, ICM is closer to the component and connector view of the architecture than CSM and KLAPER. And the performance meta-model we use is close to the input needed by evaluation procedures based on RMA. An important contribution of our reasoning framework is the interpretation, which transforms the intermediate model into a performance model with simple semantics that can be analyzed by different procedures, including those that do not directly support rich semantics such as forking, joining, and locking.

D'Ambrogio [15] describes a framework to automate the building of performance models from UML design models. The approach uses meta-models to represent the abstract syntax of source and target models and then describe the transformation from one to the other using a model transformation language. This approach does not use intermediate models to reduce the semantic distance between source and target models.

Gilmore and Kloul [18] do performance modeling and prediction from UML models that include performance information in the transition labels of the state diagrams. They use performance evaluation process algebra (PEPA) [19] as an intermediate representation of the model. A key difference with our work is that our reasoning framework focuses on fixed-priority preemptive scheduling, making it suitable to analyze hard real-time systems. PEPA, on the other hand,

assumes activities with exponentially distributed duration, whose memoryless property allows to treat preemption-resume scenarios as preemption-restart with resampling [20]. This approach is not suitable for real-time systems where more determinism is required.

Becker et al. [21] use the Palladio Component Model (PCM) to model component-based architectures including the information necessary for performance prediction. PCM is much more detailed than the ICM. For instance, component interfaces are first-class elements of the metamodel because they are used to check whether the connections between components through required and provided interfaces are valid. The elements closest to interfaces in ICM are sink and source pins. They are not associated with a type or service signature because it is assumed that the validity of the connection has already been established at the architecture description level–the CCL specification in our case. PCM also allows modeling the behavior of the component as far as necessary–an approach called gray-box–to determine the way required services and resources are used. This includes modeling parameter dependencies, loops, and branching probabilities. In ICM all the required services are assumed to be used exactly once for every invocation of the component. Certainly, compared to ICM, PCM allows modeling more details, that in turn means making fewer assumptions. However, to the best of our knowledge, the complete details in PCM models cannot be automatically generated and PCM models can only be evaluated by simulation. In contrast ICM models are automatically generated from architecture descriptions and can be analyzed not only by simulation but also by a sound performance theory for worst-case response time and schedulability. Also, the simulation framework used with PCM does not support priority-based preemptive scheduling.

Analytic constraints play an important role in our approach because they define the space of designs that are analyzable by the reasoning framework. This characteristic is also present in the work of Gherbi and Khendek [16] where OCL is used to specify the constraints and assumptions of the schedulability analysis.

8 Conclusions

We began to work in the performance reasoning framework circa 2001. The initial versions had limited analysis capability, but successful validation of the predictions revealed great potential. More recently, we have expanded the space of analyzable systems by incorporating and adapting performance theories and diversifying the set of tools used in the evaluation procedure. In this process we found that creating metamodels and model transformations greatly reduced complexity in the reasoning framework implementation.

The performance reasoning framework has been applied successfully in several industry scenarios [22–24] and has proven to be very useful for early adopters of this technology. In maintenance scenarios, model-driven analysis is also useful. The performance annotations of the components in the architecture description can be changed to reflect the intended modifications and a new run of the analysis can verify whether the modifications will yield the required performance.

The performance reasoning framework is packaged as an Eclipse plug-in and can be used with different design languages thanks to the ICM metamodel and design-to-ICM adapters. A simple architecture description with structural information (wiring of components and connectors through synchronous and asynchronous ports) and performance annotations (e.g., priority, execution time) is the input for performance analysis. ArchE [25], an architecture expert tool, is an example of an Eclipse-based tool that has been adapted to use the reasoning framework. The PSK is a fully automated solution that includes the performance reasoning framework and uses CCL as the design language, providing a comprehensive MDE solution: the same architecture description enhanced with behavior information can also be used as input for code generation. An important consequence is that conformance between the code, the architecture and the analysis results is maintained.

Architecture description languages that have explicitly considered the characteristics of an application domain or the business needs of adopting organizations have been more successful [26]. The CCL language was designed to support the development of component-based safety-critical real-time systems, and its semantics are close to the target runtime environment. As a result, annotations that express the properties of components and connectors are simpler—by contrast, a UML generic component or assembly connector would require extensive stereotyping and far more annotations to express the same things.

The performance reasoning framework continues to evolve. Working with academic and industry collaborators, we plan to extend the space of analyzable systems by relaxing some of the current analytic constraints. Integration with other performance analysis tools is also a goal. A limitation of our performance reasoning framework is that execution time variations caused by branching are only represented by the resulting execution time distributions. Since CCL and other design languages can express the behavior inside components, future work intends to overcome that limitation by having the interpretation look inside the state machine of the components, perhaps using a gray-box approach as in PCM, but trying not to hinder the automation or ability to analyze the model by means other than simulation.

References

1. Schmidt, D.: Model-driven engineering. IEEE Computer Magazine **39**(2) (2006)
2. Ivers, J., Moreno, G.A.: Model-driven development with predictable quality. In: Companion to the OOPSLA'07 Conference. (2007)
3. Bass, L., Ivers, J., Klein, M., Merson, P.: Reasoning frameworks. Technical Report CMU/SEI-2005-TR-007, Software Engineering Institute (2005)
4. Klein, M.H., Ralya, T., Pollak, B., Obenza, R., Gonzalez Harbour, M.: A practitioner's handbook for real-time analysis. Kluwer Academic Publishers (1993)
5. Hissam, S., Klein, M., Lehoczky, J., Merson, P., Moreno, G., Wallnau, K.: Performance property theories for predictable assembly from certifiable components (PACC). Technical Report CMU/SEI-2004-TR-017, Software Engineering Institute (2004)

6. Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison Wesley (2003)
7. Wallnau, K., Ivers, J.: Snapshot of CCL: A language for predictable assembly. Technical Note CMU/SEI-2003-TN-025, Software Engineering Institute (2003)
8. Gonzalez Harbour, M., Klein, M., Lehoczky, J.: Timing analysis for fixed-priority scheduling of hard real-time systems. IEEE Trans. Softw. Eng. **20**(1) (1994)
9. Sprunt, B., Sha, L., Lehoczky, J.: Aperiodic task scheduling for hard real-time systems. Real-Time Systems **1**(1) (1989)
10. Hissam, S., Moreno, G., Stafford, J., Wallnau, K.: Enabling predictable assembly. Journal of Systems and Software **65**(3) (2003) 185–198
11. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005. (2005)
12. Gonzalez Harbour, M., Gutierrez Garcia, J.J., Palencia Gutierrez, J.C., Drake Moyano, J.M.: MAST: Modeling and analysis suite for real time applications. In: The 13th Euromicro Conference on Real-Time Systems. (2001)
13. Krahl, D.: Extend: the Extend simulation environment. In: WSC '02: Proceedings of the 34th Winter Simulation Conference. (2002)
14. Grassi, V., Mirandola, R., Sabetta, A.: From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In: 5th International Workshop on Software and Performance. (2005)
15. D'Ambrogio, A.: A model transformation framework for the automated building of performance models from UML models. In: 5th International Workshop on Software and Performance. (2005)
16. Gherbi, A., Khendek, F.: From UML/SPT models to schedulability analysis: a metamodel-based transformation. In: Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing. (2006)
17. Woodside, M., Petriu, D., Petriu, D., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: 5th International Workshop on Software and Performance. (2005)
18. Gilmore, S., Leila, K.: A unified tool for performance modelling and prediction. Reliability Engineering and System Safety **89(1)** (Jan 2005) 17–32
19. Hillston, J.: Tuning systems: From composition to performance. The Computer Journal **48**(4) (May 2005) 385–400 The Needham Lecture paper.
20. Gilmore, S., Hillston, J.: The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In: Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Number 794 in Lecture Notes in Computer Science, Vienna, Springer-Verlag (May 1994) 353–368
21. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: WOSP '07: Proceedings of the 6th International Workshop on Software and Performance, New York, NY, USA, ACM (2007) 54–65
22. Hissam, S., Hudak, J., Ivers, J., Klein, M., Larsson, M., Moreno, G., Northrop, L., Plakosh, D., Stafford, J., Wallnau, K., Wood, W.: Predictable assembly of substation automation systems: An experiment report, second edition. Technical Report CMU/SEI-2002-TR-031, Software Engineering Institute (2003)
23. Larsson, M., Wall, A., Wallnau, K.: Predictable assembly: The crystal ball to software. ABB Review (2) (2005) 49–54
24. Hissam, S., Moreno, G.A., Plakosh, D., Savo, I., Stelmarczyk, M.: Predicting the behavior of a highly configurable component based real- time system. In: ECRTS '08: Proceedings of the 20th Euromicro Conference on Real-Time Systems, IEEE Computer Society (2008)

25. Bachmann, F., Bass, L.J., Klein, M., Shelton, C.P.: Experience using an expert system to assist an architect in designing for modifiability. In: 4th Working IEEE/IFIP Conference on Software Architecture (WICSA). (2004)
26. Medvidovic, N.: Moving architectural description from under the technology lamppost. In: 32nd Euromicro Conference on Software Engineering and Advanced Applications. (2006)