June, 2008

Performance Analysis of Real-Time Component Architectures: A Model Interchange Approach

Gabriel A. Moreno, *Software Engineering Institute*
Connie U. Smith
Lloyd G. Williams

Performance Analysis of Real-Time Component Architectures: A Model Interchange Approach

Gabriel A. Moreno
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
+1.412.268.1213
gmoreno@sei.cmu.edu

Connie U. Smith
Performance Engineering Services
PO Box 2640
Santa Fe, NM 87504
+1.505.988.3811
www.spe-ed.com

Lloyd G. Williams
PerfX
2345 Dogwood Circle
Louisville, CO 80027
+1.720.890.8116
lloydw@perfx.net

ABSTRACT
Model interchange approaches support the analysis of software architecture and design by enabling a variety of tools to automatically exchange performance models using a common schema. This paper builds on one of those interchange formats, the Software Performance Model Interchange Format (S-PMIF), and extends it to support the performance analysis of real-time systems. Specifically, it addresses real-time system designs expressed in the Construction and Composition Language (CCL) and their transformation into the S-PMIF for additional performance analyses. This paper defines extensions and changes to the S-PMIF meta-model and schema required for real-time systems. It describes transformations for both simple, best-case models and more detailed models of concurrency and synchronization. A case study demonstrates the techniques and compares performance results from several analyses.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Modeling techniques; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.12 [**Software Engineering**]: Interoperability; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms
Performance, Design

Keywords
Performance, software performance engineering, performance model, performance analysis, model interchange, real-time systems, architecture analysis, component-based systems

1. INTRODUCTION
Performance is a quality attribute that, in spite of being critical to a large number of software systems, is often not appropriately addressed. As a result, many software-based systems fail to meet their performance requirements as implemented. Fixing performance problems often causes cost and schedule overruns and, in some cases, the software cannot be fixed and must be abandoned.

Performance cannot be retrofitted; it must be designed into software from the beginning. Our experience is that performance problems are most often due to inappropriate architectural choices rather than inefficient coding. By the time the architecture is fixed, it may be too late to achieve adequate performance by tuning. Thus, it is important to be able to assess the impact of architectural decisions on quality requirements such as performance and reliability at the time that they are made.

Although sound performance analysis theories and techniques exist, they are not widely used because they are difficult to understand and require heavy modeling effort throughout the development process [1]. Consequently, software engineers usually resort to testing to determine whether the performance requirements have been satisfied. To ensure that these theories and techniques are used, they must be made more accessible—integrated into the software development process and supported with tools.

This paper illustrates an approach to making performance analysis more accessible. It makes several contributions:

- Demonstrates the use of standard performance modeling techniques for component-based real-time systems
- Illustrates the use of the Software Performance Model Interchange Format (S-PMIF) with the Construction and Composition Language (CCL)
- Merges streams of research that have thus far been independent: predictable assembly of components, software performance engineering, and model interchange.

The next section provides some background on the merged streams of research, and then Section 3 discusses related work in these areas. Section 4 provides an overview of the Construction and Composition Language (CCL) and the ICM meta-model for CCL assemblies. Next, Section 5 presents the revised S-PMIF meta-model for real-time systems. Section 6 describes the implementation of the interoperability features. Section 7 presents a case study as proof of concept and Section 8 offers some conclusions.

WOSP'08, June 24–26, 2008, Princeton, New Jersey, USA.

2. BACKGROUND

As noted above, this work merges several distinct streams of research. This section describes these streams and provides an overview of their merger.

2.1 Predictable Assembly

The research on predictable assembly focuses on the development of technologies and methods to enable the development of software with predictable runtime behavior [2-4]. The PACC initiative at the Software Engineering Institute proposes the use of smart constraints to achieve predictability by construction [5]. The idea behind this concept is that analysis theories rely on certain assumptions in order to be applicable, which means that the behavior of a software system is predictable by a given theory only if it satisfies its assumptions. Smart constraints can guarantee the satisfaction of these assumptions so that if a software system can be constructed under these constraints, then its behavior can be predicted. Smart constraints can be enforced by different means, from automated checks at the architecture description level or design specification to imposition through component containers [6, 7].

Evaluation is as important as smart constraints in order to achieve predictability by construction. Since the complexity of performance evaluation and the effort required for creating the performance models has been cited as one of the root causes of software performance failures, it is critical to automate them to provide a solution to this recurring problem. One way of doing so is by using reasoning frameworks [8]. A reasoning framework encapsulates an analysis theory, the generation of theory specific models from the architecture or design specification, and the evaluation of these models.

All these concepts of predictable assembly have been integrated together and demonstrated in the PACC Starter Kit (PSK) [9]. The PSK is a development environment that includes the Construction and Composition Language (CCL) [10], a language to describe the interface and behavioral specification of components and their assembly into systems. The runtime behavior of these systems specified in CCL can be predicted with the performance and model checking reasoning frameworks. Furthermore, executable code targeting the included runtime environment (the Pin component technology [11] and a real-time extension for Windows) can be generated from the same specification, guaranteeing that the code matches the specification. All the technologies integrated in this model-driven approach allow making performance predictions throughout the development lifecycle, from the early stages in which only the component and connector view of the architecture and execution time estimates are available, to the point in which executable code can be generated from the behavioral specification and measured. It even allows predicting the impact of changes during maintenance.

Although the architecture of the PSK allows the integration of third-party performance analysis tools via plug-ins [12], the integration of each new tool requires the development of a new transformation to generate a performance model in an input format suitable for the tool. Even though this approach provides tight integration and allows exploiting specific features of the different tools, another promising option is the tool interoperability approach using an interchange format [13]. This paper describes the use of the Software Performance Model Interchange Format (S-PMIF) [14, 15] to allow the analysis of real-time designs specified in CCL with additional performance analysis tools.

2.2 Software Performance Engineering

Software performance engineering (SPE) is a systematic, quantitative approach to constructing software systems that meet performance requirements. SPE prescribes principles for creating responsive software, the data required for evaluation, procedures for obtaining performance specifications, and guidelines for the types of evaluation to be conducted at each development stage. It incorporates models for representing and predicting performance as well as a set of analysis methods [16].

SPE advocates three modeling strategies:

1. *Simple-model strategy*: Start with the simplest possible model that identifies problems with the system architecture, design, or implementation plans.
2. *Best- and Worst-Case Strategy*: Use best- and worst-case estimates of resource requirements to establish bounds on expected performance and manage uncertainty in estimates.
3. *Adapt-to-Precision Strategy*: Match the details represented in the models to the knowledge of the software processing details.

Simple models are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance requirements. As the software process proceeds, the models are refined to more closely represent the performance of the emerging software (adapt to precision strategy). If the predicted best-case performance is unsatisfactory, developers seek feasible alternatives. If the worst- case prediction is satisfactory, they proceed to the next step of the development process. If the results are somewhere in-between, analyses identify critical components and seek more precise data for them. A variety of techniques can provide more precision, including: further refining the architecture and constructing more detailed models or constructing performance prototypes and measuring resource requirements for key components.

SPE·ED [17] is a tool designed specifically to support the SPE methods and models defined in [16]. Using a small amount of data about envisioned software processing, *SPE·ED* creates and solves performance models, and presents visual results. It provides performance data for requirements and design choices and facilitates comparison of software and hardware alternatives for solving performance problems.

SPE·ED supports four types of solutions for the performance models:

1. No contention – analytic solution with one user
2. Contention – analytic solution of multiple users of the same scenario,
3. System model – simulation solution of all scenarios and users
4. Advanced model – analysis of communication and coordination among scenarios and users.

The simple model solution (no contention) suffices for most performance analyses early in development. The data that is available at that time usually doesn't provide the precision needed for the more detailed solutions. Later, the advanced system model solution gives more insight into situations when mean values may be fine, but queue lengths may build in some circumstances and lead to unacceptable performance. The advanced system model executes the simulation and actually "makes calls" to other processes at the point in the execution where special synchronization nodes are placed. If the called process is busy, the calling process waits in a queue.

In *SPE•ED*, an advanced system execution model is automatically created and solved to quantify contention effects and delays.

2.3 Model Interchange

Model interchange seeks cooperation among existing tools that perform different tasks. XML-based interchange formats for models provide a mechanism whereby model information may be transferred among modeling and analysis tools. This makes it possible for a user to create a model in one tool, perform some studies, and then move the model to another tool for other studies that are better done in the second tool.

The Software Performance Model Interchange Format (S-PMIF) [14] is a common representation that can be used to exchange information between software design tools and software performance engineering tools. With S-PMIF, a software tool can capture software architecture and design information along with some performance information and export it to a software performance engineering tool for model elaboration and solution without the need for laborious manual translation from one tool's representation to another, and without the need to validate the resulting specification. Use of the S-PMIF does not require tools to know about each other's capabilities, internal data formats, or even existence. It requires only that the importing and exporting tools either support the S-PMIF or provide an interface that reads/writes model specifications from/to a file.

S-PMIF enables the following SPE tasks:

1. Developers can prepare designs as usual and export the data to SPE tools where performance models can be constructed automatically.
2. The model transformation can be used to check that the resulting processing details are those intended by the software specification.
3. Data available to developers can be captured in the development tool – other data can be added by performance specialists in the SPE tool.
4. Rapid production of models makes data available for supporting design decisions in a timely fashion. This is good for studying architecture and design tradeoffs before committing to code.
5. Developers can create and evaluate some SPE models without needing detailed knowledge of performance models.

The performance model interchange formats specify the model and a set of parameters for one run. For model studies, however, it is useful to be able to specify multiple runs, or experiments, for the model. In [18] an XML interchange schema extension, called Experiment Schema Extension (Ex-SE), defines a set of model runs and the output desired from them. This extension to an interchange schema provides a means of specifying performance studies that is independent of a given tool paradigm.

Thus, the model interchange approach makes it possible to create a software specification in a development tool, then automatically export the model description and some specifications for conducting performance assessments, and obtain the results for use in considering architectural and design alternatives. The advantages of this approach are: it is relatively easy to accomplish with existing tools; it requires minor extensions to tool functions (import and export) or creation of an external translator to convert file formats to/from interchange formats; and it enables the use of multiple tools so it is easy to compare results and to use the tool best suited to the task.

Without a shared interchange format, two tools would need to develop a custom import and export mechanism. A third tool would require a custom interface between each of those tools resulting in a $4 \cdot (N! / (2!(N-2)!))$ requirement for customized interfaces. With a shared interchange format, the requirement for customized interfaces is reduced to $2 \cdot N$. With XML tools the complexity and amount of effort to create the interface is quite small [19]. While XML is verbose, model interchange is a coarse-grained interface. A file is exported, sent to another tool, it is imported and the model solved. So the performance impact of XML as the interface is insignificant compared to a fine-grained interface that exchanges each XML element as it is generated.

3. RELATED WORK

3.1 Architecture Assessment

Kazman and co-workers describe two related approaches to the evaluation of software architectures. The Software Architecture Analysis Method (SAAM) [20] uses scenarios to derive information about an architecture's ability to meet certain quality requirements such as performance, reliability, or modifiability. The Architecture Tradeoff Analysis Method (ATAM) [21] extends SAAM to consider interactions among quality requirements and identify architectural features that are sensitive to more than one quality attribute. Once these sensitivities have been identified, tradeoffs between quality requirements can be evaluated.

PASA^{SM} [22] is a method for the performance assessment of software architectures. It uses the principles and techniques of SPE [16] to identify potential areas of risk within the architecture with respect to performance and quality objectives. If a problem is found, PASA also identifies strategies for reducing or eliminating those risks. PASA is similar to SAAM and ATAM in that it is scenario-based. However, there are also important differences. In SAAM and ATAM, scenarios are informal narratives of uses of the software. In PASA, performance scenarios are expressed formally using UML sequence or activity diagrams. ATAM and PASA differ in their approach to performance modeling. ATAM uses analytical models of certain architectural features while PASA uses more general software execution and system execution models that may be solved analytically or via simulation [16]. Both SAAM and ATAM produce a list of problem areas or risks while PASA produces a quantitative estimate of the performance of the system as implemented as well as for proposed changes. Finally, ATAM is also concerned with interactions between

quality attributes and focuses on architectural features where tradeoffs may be required. While PASA's primary focus is on performance, quality attributes and tradeoffs between them are considered as well.

Earlier approaches to architecture assessment (e.g., [23], [24] [25], [26], [27], and [28]) relied on directly connecting a particular design notation and a particular type of performance model. More recently, interchange formats have been used to decouple the architecture description from the model description (see below).

3.2 Model Interchange

Several model interchange formats for different types of models have been proposed. The Performance Model Interchange Format, PMIF, [13, 29] enables various tools to exchange queueing network model information. PMIF is based on a meta-model, which provides an underlying formalism for the schema. The meta-model for the Software Performance Model Interchange Format, S-PMIF, was defined and later extended in [15, 30]. It differs from the PMIF in that it specifies *software* processing details and bridges the gap between software architecture and design tools and performance analysis tools. Woodside et al. developed a meta-model, PUMA, that combines software and system models based on layered queueing networks (LQN) in [31]. D'Ambrogio also defines a MOF meta-model of LQNs and transfers UML models to LQNs in [32].

Other approaches have focused on transferring information between UML-based software design tools and software performance engineering tools, such as [14, 33-35]. Gu and Petriu [36] and Balsamo and Marzolla [24] use XML to transfer design specifications into a particular solver; however, they do not attempt to develop a general format for the interchange of performance models among different tools. Our work does not involve UML transformations so other topics such as SPT and MARTE are not addressed here.

This body of work demonstrates that model interoperability among a set of tools is viable. Common interchange formats such as PMIF, S-PMIF, and PUMA are preferable because they enable the use of a large number of tools without requiring custom interfaces for each one.

3.3 Component-Based Approaches

Some work has addressed the performance analysis of component-based systems. Wu and Woodside use an XML Schema to describe the contents and data types that a Component-Based Modeling language (CBML) document may have [37]. CBML is an extended version of the Layered Queuing Network (LQN) language that adds the capability and flexibility to model software components and component-based systems.

Becker, et al., address components whose performance behavior depends on the context in which they are used [38]. They address sources of variability such as loop iterations, branch conditions, and parametric resource demand, and then use simulation to predict performance in a particular usage context.

Grassi, et al., extend the KLAPER MOF meta-model to represent reconfigurable component-based systems in [39]. It is to be used in autonomic systems and enable dynamic reconfiguration to meet QoS goals.

These approaches are performance-centric in that they create/adapt a model of component based systems specifically for performance assessment. We prefer to work with generally accepted architecture representations, and use a common interchange format (S-PMIF) that allows the use of a variety of performance modeling tools to provide performance predictions for architecture and design alternatives. In addition, we have extended the S-PMIF to include features necessary for evaluating real-time systems. In the future, it may be possible to unify the various interchange formats as suggested by [40]. In the meantime, it makes sense to extend the meta-models as necessary to create a superset of the necessary information for performance assessment.

4. CCL AND ICM

The architecture specification language used in this study is the Construction and Composition Language (CCL) [10]. This section describes relevant features of CCL and ICM, a meta-model for facilitating the analysis of CCL specifications.

4.1 Construction and Composition Language

CCL is a language for specifying the behavior of components, their composition to form assemblies or systems, and the properties required for reasoning about the assemblies [10]. CCL enforces the notion of pure composition, which means that all the behavior is inside the components and systems are assembled by wiring components together with no "glue" code. Components in CCL interact through *pins*. Source pins emit stimuli and sink pins receive stimuli. When a sink pin receives a stimulus, it triggers a *reaction*, which carries out the response to the stimulus. A reaction can initiate an interaction with other components via its source pins. Pins can interact synchronously or asynchronously. Stimuli can carry data, and for that reason, pins have signatures describing the data they consume and produce.

The following CCL specification declares a component type *MovementPlanner* with one asynchronous sink pin and three source pins (one synchronous and two asynchronous). Then it declares a reaction in which all the pins participate, that is, it is triggered by *go*, the only sink pin, and it can interact with other components through the source pins. The keyword *threaded* indicates that this reaction executes in its own thread.

```
component MovementPlanner() {
    sink asynch go();
    source synch get(produce int mode, produce
        string in, consume string out);
    source asynch moveX(produce int pos);
    source asynch moveY(produce int pos);

    threaded react reaction go, get, moveX, moveY)
    {
        // reaction specification goes here
    }
}
```

It is important to note that a specification like this that does not have the behavioral specification of the reaction is a valid CCL specification. Therefore, analysis can be done in the early stages of the design, when only the component and connector structure of the system is known.

An assembly of components is produced by creating component instances and connecting them as in the following fragment.

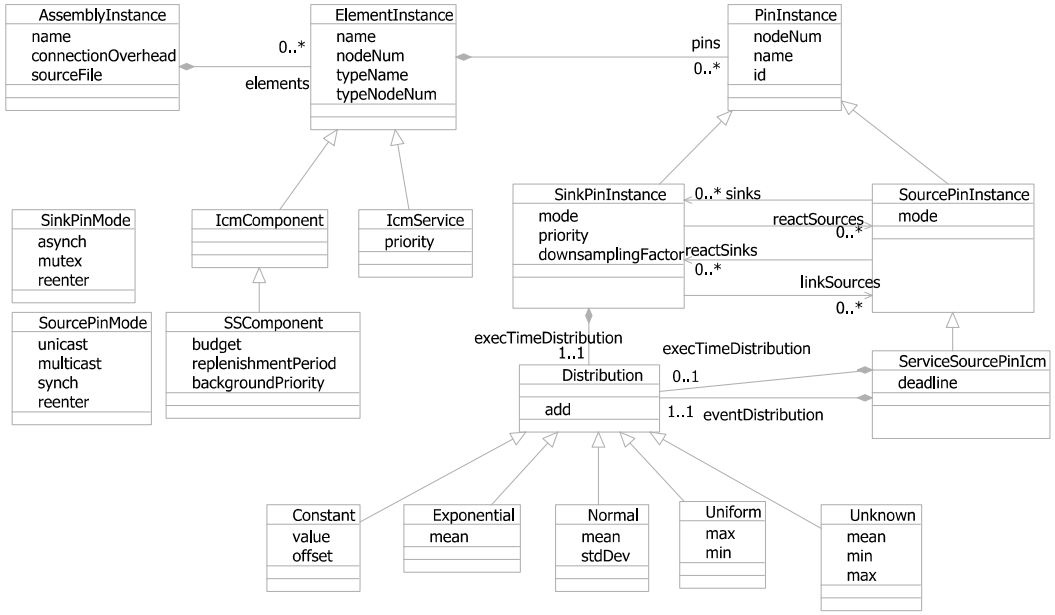

Figure 1. ICM Meta-model

```
MovementPlanner movementPlanner();
AxisController controllerX("X");

movementPlanner:moveX ~> controllerX:move;
```

For the connection between two pins to be legal, they need to have the same mode (synchronous or asynchronous) and they need to have complementing signatures, meaning that the data produced by one pin is consumed by the other and vice versa. For example, the signature of the pin *move* in *AxisController* is as follows.

```
sink asynch move(consume int pos);
```

Assemblies declare *services* (e.g., clocks, keyboard input, console output, etc.) that they expect the environment to provide. The specification of a service is identical to that of a component, except that the keyword *service* is used instead. One important semantic difference though, is that services are the only source of external events because components cannot interact directly with the environment.

CCL has an annotation mechanism that can be used to provide information required to analyze the assembly. For example, the following annotation[1] indicates the minimum, average, and maximum execution time of the *move* pin in *AxisController* when run in isolation (i.e., with no blocking and no preemption).

```
annotate AxisController:move {"lambda*",
    const string execTime =
        "G(9.95, 10.01, 10.14)" }
```

Only the aspects of CCL most relevant for this paper have been covered here. More details about CCL can be found in [10].

4.2 ICM: A Meta-model for CCL Assemblies

The intermediate constructive model (ICM) is an intermediate representation of a CCL assembly that makes the generation of analysis models simpler. Instead of having to deal with the language related constructs in the CCL abstract syntax tree while developing a transformation, it is easier to start from concepts that are more relevant to reason about the assembly. For example, it is easier to reason about a source pin with an event interarrival distribution, than doing the same thinking in terms of a computational unit, an annotation and a float literal expression.

The ICM meta-model, shown in Figure 1, does not have information regarding types and only represents instances. That is, if there are two instances of the same component type, elements common to both, such as pins, are repeated in the model. This redundancy also makes it easier to traverse the design in order to transform it to an analysis model. The root element of the ICM meta-model is the *AssemblyInstance*, which contains all the service and component instances in the assembly. These have a common base class, *ElementInstance*, with all the attributes they share. Components and services have pins that can be either sink or source. *SinkPinInstance* has an execution time distribution to represent the amount of CPU time the sink pin requires. When a source pin belongs to a service (i.e., it is a *ServiceSourcePinIcm*), it has an event interarrival distribution and can optionally have an execution time distribution as well. Distributions can be of different kinds, such as constant or exponential. In order to represent the connections between components, there is a reference *sinks* between pins that shows which sink pins are connected to a source pin. In a similar way, the *reactSources* reference indicates the sources that are triggered by a sink pin in the same component.

5. S-PMIF

The S-PMIF is based on the SPE meta-model. This meta-model defines the essential information required to create the software and system performance models as defined in [16]. The SPE

[1] The argument "lambda*" indicates the reasoning framework this annotation is used for.

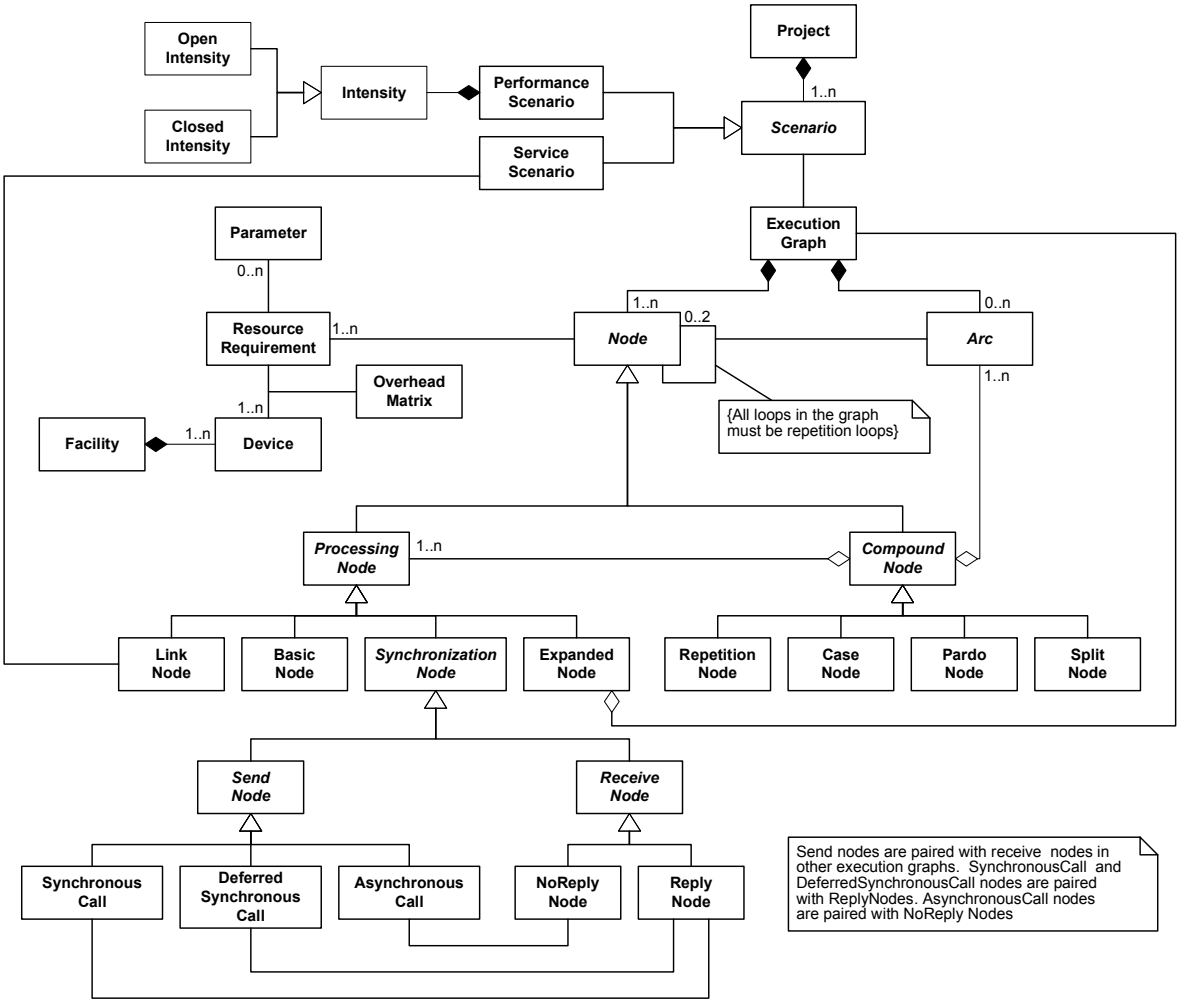

Figure 2. S-PMIF Meta-Model

meta-model class diagram is shown in Figure 2. The complete definition is available at www.spe-ed.com/pmif/s-pmif.xml [2]

Several changes were made to the meta-model described in [30] as a result of this work. The first was the creation of the abstract entity Scenario with subclasses PerformanceScenario and ServiceScenario. A PerformanceScenario represents an end-to-end, externally visible interaction (analogous to a Use Case) while a ServiceScenario is a scenario that provides one or more services to one or more PerformanceScenarios. Performance Scenarios have workload intensities which may be specified by a number of users and think time (closed workload) or an inter-arrival time (open workload). ServiceScenarios have an optional intearrivalTime (default is 0) and numberOfInstances.

Several attributes were also added to the meta-model to allow specification of real-time concepts:

- arrivalDistribution (PerformanceScenario) and serviceDistribution (Device). These take their values from an enumerated type, DistributionType (exp, normal, constant, erlang, hyperexp, uniform(u1,u2)). These attributes are optional (default to exp).
- schedulingDiscipline (Device). This attribute is also an enumerated type (FCFS, IS, LCFSPR, PR, PS, RR) and is optional.
- responseTimeRequirement and throughputRequirement (Scenario). The values of these attributes are real numbers.

In addition, the attributes partnerNodeID and partnerScenarioName were added to SendNode and attributes were removed from SynchronizationNode.

TheS-PMIF is implemented using three separate schemas: *Topology*, *OverheadMatrix*, and *Device*. They can be combined by including the appropriate schemas. Thus, *Topology* may include *OverheadMatrix* which includes *Device*. This is useful because one may use any of the schemas without using the others. For example, if the overhead matrix specification is coming from another source it does not need to be included in the topology, and vice-versa.

[2] You also need ~/OverheadMatrix.xml and ~/Devices.xml. The extension for all schemas is ~.xml so that it can be viewed from a browser. Change the extensions to ~.xsd to use them.

Comparing this meta-model to the MARTE specification is beyond the scope of this paper and will be addressed in future work.

6. IMPLEMENTATION

6.1 Generating S-PMIF Models from CCL

Even though from the user's perspective the transformation to an S-PMIF model starts from a CCL specification, behind the scenes the CCL specification is transformed first to an ICM model from which the S-PMIF is finally generated.

The ICM meta-model is defined as an Ecore model, the meta-model of the Eclipse Modeling Framework [41]. EMF can generate the Java implementation classes to load, manipulate and persist instances of the model. The S-PMIF format is specified as an XML schema, and since EMF provides the same generative capabilities starting with an XML schema, EMF was used to generate the Java implementation to manipulate the S-PMIF models.

The following sections describe the generation of two flavors of S-PMIF model from ICM, the simple model, or no contention model, and the advanced model.

6.2 Generation of the Simple Model

The overall approach to generate the simple model consists of creating an S-PMIF performance scenario for each service source pin in the ICM. In that way, the performance scenario encompasses the complete response to an external event. The execution graph for the performance scenario is created by recursively traversing the response by visiting each pin, starting with the service source pin. Figure 3 shows the pseudocode for the two functions that implement the core of the transformation. The function *visitSource* checks whether the source pin is synchronous or asynchronous. In the first case, it directly returns the node that is created by visiting the sink connected to that source. However, if the source pin is asynchronous, it creates a *SplitNode* to represent the initiation of concurrent threads of execution, and adds to the split node the nodes resulting from visiting all the sink pins connected to the source node. The function *visitSink* creates a *BasicNode* with a *ResourceRequirement* to model the computation carried out by the sink pin and then it visits in sequence all the source pins in the same component that are triggered by the reaction of the sink pin. The order of execution is modeled by creating the arcs connecting the nodes.

```
Node visitSource(SourcePinInstance source) {
  if source is synchronous {
    node = visitSink(sink)
  } else { // source is asynchronous
    node = new SplitNode
    for each sink in source.sinks {
      newNode = visitSink(sink)
      add newNode to children of node
    }
  }
  return node
}

Node visitSink(SinkPinInstance sink) {
  node = new BasicNode
  add ResourceRequirement to node from
        sink.execTimeDistribution
  lastNode = node
  for each source in sink.reactSources {
    newNode = visitSource(source)
    arc = new Arc
    arc.from = lastNode.getNodeId()
    arc.to = newNode.getNodeId()
    lastNode = newNode
  }
  return node
}
```

Figure 3. Pseudocode for simple S-PMIF model generation

One problem that arose while implementing this algorithm was the lack of subtype relationships between the different kinds of nodes in the S-PMIF schema. In the S-PMIF meta-model, both *BasicNode* and *SplitNode* are subtypes of *Node*. However, in the XML schema for S-PMIF the hierarchy was flattened and those relationships were lost [14]. For that reason, in the Java implementation generated with EMF from the S-PMIF schema, *Node*, *BasicNode*, and *SplitNode* have no subtype relationship. This complicates the implementation of the transformation algorithm. For instance, what is the return type of *visitSource* if it can return either a *BasicNode* or a *SplitNode*? The problem also hindered the use of polymorphism because it makes it impossible to make calls such as *lastNode.getNodeId(),* where *lastNode* can refer to different types of nodes. Although the intent of flattening the S-PMIF schema was to simplify the XML [14], the lack of subtype relationships proved to have the opposite effect in situations where the XML is generated by a high level modeling technology such as EMF.

The problem of not having node subtyping was overcome in two different ways. One solution was changing the return type of *visitSource* and *visitSink* to *ExpandedNode*, and wrapping the result of each function in its own execution graph contained in an expanded node. This approached worked well albeit it generated a lot of expanded nodes and execution graphs that would otherwise not be needed.

The second solution was more complicated because it consisted of adding subtyping to the schema from which the Java implementation classes were generated while maintaining an output format compliant with the original S-PMIF schema. The subtyping was added by using the schema type extension mechanism. In addition, containment relationships that were implemented with XSD choice were changed to use the base type. For example, the containment relationship shown in Figure 4 was changed as it appears in Figure 5. This change allowed EMF to generate Java code with the right subtype relationships. However, the generated XML for a *BasicNode* would look as follows.

```
<Node xsi:type="BasicNode_type" NodeId="N1" … />
```

Since this is not compatible with the S-PMIF schema, XSD substitution groups were defined so that the desired XML output

```
<xs:complexType name="EG_type">
  <xs:sequence>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="BasicNode"
                  type="BasicNode_type"/>
      <xs:element name="SplitNode"
                  type="CPSNode_type"/>
      ...
    </xs:choice>
  </xs:sequence>
  ...
</xs:complexType>
```

Figure 4. Containment with schema choice

was produced. A substitution group introduced to the schema with

```
<xs:element name="BasicNode"
  substitutionGroup="Node" type="BasicNode_type"/>
```

resulted in the right XML produced as in this example:

```
  <BasicNode NodeId="N1" … />
```

```
<xs:complexType name="EG_type">
  <xs:sequence>
    <xs:element maxOccurs="unbounded"
                name="Node" type="Node_type"/>
    ...
  </xs:sequence>
  ...
</xs:complexType>
```

Figure 5. Containment with base type

6.3 Generation of the Advanced Model

In a component-based real-time system, the response to an event may be realized by several components that may execute in their own thread. When creating the advanced S-PMIF model, the different concurrent threads of execution need to be modeled so that contention between them can be evaluated.

S-PMIF has the concept of a *SynchronizationNode* that maps directly to the different kinds of pins in CCL. Synchronous source and sink pins can be represented by *SynchronousCall* and *Reply* nodes respectively. Asynchronous source and sink pins can be modeled by *AsynchronousCall* and *NoReply* nodes correspondingly. The pseudocode for the algorithm used to generate the advanced model is shown in Figure 6. The most important function is *getPSForSink*. This function creates a scenario for a sink pin in the assembly only if it has not created it before; otherwise, it returns the already existing scenario. The performance scenario starts with either a *BasicNode* or *SynchronizationNode* depending on whether it is top level (i.e., first in the response to an event) or not. If it is not top level, the type of the *SynchronizationNode* is set to match the interaction mode of the pin. This first node in the scenario has a *ResourceRequirement* specifying the execution time required by the sink pin in the CPU. If the component interacts with other components via its source pins, synchronization nodes of type *SynchronousCall* or *AsynchronousCall* are created to model the interactions with the connected sink pins. In order to get the partner scenario of these synchronization nodes, *getPSForSink* is called recursively. The main function of the transformation, *generateModel*, just calls *getPSForSink* for each of the sinks connected to service source pins in the assembly and sets the corresponding interarrival time for the top level performance scenarios.

The algorithm presented here depends on a simplifying assumption, namely, that all the sink pins in the assembly participate in threaded reactions. Nevertheless, it would not be difficult to extend it to support unthreaded reactions as well because traversing unthreaded reactions would be the same as was done in the simple model generation algorithm, except that in this case there would be no split nodes.

6.4 Importing the Models

The S-PMIF is imported into a software performance modeling tool, like *SPE·ED* [42, 43], SP[44], or HIT [45] for performance analysis of the software architecture and design, and evaluation of alternatives. The software performance modeling tool must either provide an import mechanism for S-PMIF or read input from a file that can be generated from a translation of the S-PMIF.

```
generateModel() {
  for each serviceSourcePin in assembly {
    linkedSink =
           sink connected to serviceSourcePin
    ps = getPSForSink(linkedSink, true)
    ps.interarrivalTime =
      serviceSourcePin.eventDistribution.mean
  }
}

PS getPSForSink(SinkPinInstance sink,
                bool topLevel) {
  if PS already created for sink {
    return psMap[sink]
  }
  ps = new PS
  ps.priority = sink.priority
  if topLevel {
    node = new BasicNode
  } else {
    node = new SynchronizationNode
    if sink is synchronous {
      node.myType = Reply
    } else {
      node.myType = NoReply
    }
  }
  add ResourceRequirement to node from
        sink.execTimeDistribution
  make node first node in ps
  lastNode = node
  for each source reacting to sink {
    for each linkedSync connected to source {
      node = new SynchronizationNode
      if sink is synchronous {
        node.myType = SynchronousCall
      } else {
        node.myType = AsynchronousCall
      }
      node.partnerPerfScenario =
            getPSForSink(linkedSink, false)
      arc = new Arc
      arc.from = lastNode.getNodeId()
      arc.to = node.getNodeId()
      lastNode = node
    }
  }
  psMap[sink] = ps
  return ps
}
```

Figure 6. Pseudocode for advanced S-PMIF model generation

We use the *SPE·ED* tool. *SPE·ED* uses the Document Object Model (DOM) to import the s-pmif.xml. It first loads and parses the document, then uses DOM calls to walk through each scenario and create the corresponding nodes and arcs in *SPE·ED*. Previous work created a prototype import mechanism [30]. It included neither the import of resource requirements nor the overhead matrix so those features were added to handle these models. This was the only extension required for the simple models. The following additional features were required for the real-time extensions used in the advanced models:

- ServiceScenarios are currently mapped to performance scenarios. In the future, *SPE·ED* will support ServiceScenarios, so this is a temporary solution.

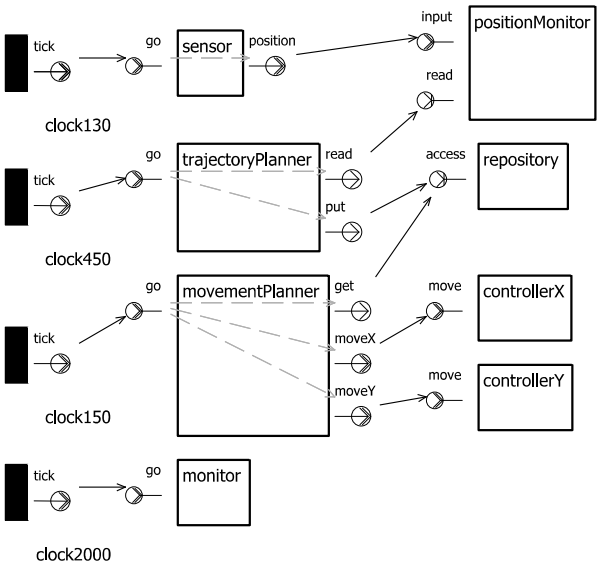

Figure 7. Robot controller design

- *SPE·ED* assumes arrival times and service times are exponentially distributed, the case study required constant interarrival and service times
- Preemptive-resume scheduling was required.
- Synchronization nodes were not supported in the earlier prototype

7. PROOF OF CONCEPT

In order to demonstrate the viability of the performance model exchange approach, we selected a real-time application that was specified with CCL. The application is a simple robot controller that takes high-level work orders for a robot and translates them to low-level movement commands for the robot's two axes. Figure 7 shows the design of the controller. The solid black boxes are sources of events, and in this case, they all have constant interarrival intervals. For clarity, the period of the event has been included in the name of the service (e.g., clock130 has a period of 130ms). Components are depicted as hollow boxes in the diagram, with sink pins on the left, and source pins on the right. Single and double arrow pins indicate synchronous and asynchronous interaction respectively.

The trajectory planner periodically receives high-level orders for the robot and, using information it gets from the position monitor, decomposes them into subwork orders, which it then puts in the work order repository. The movement planner gets orders from the repository and translates them into movement commands for the axis-controllers *controllerX* and *controllerY*. The position monitor receives input from a sensor that is read periodically, and the *monitor* component performs low-priority monitoring tasks.

It is critical that the movement planner never finds the repository empty because if it does, it has to abort the operation of the robot. Both planners cannot miss their deadline at the end of their period. Therefore, this is a hard real-time situation. All the sink pins in this design execute on their own thread at different priorities.

```
<PerformanceScenario EGId="clock450.tick"
InterarrivalTime="450.0" NumberOfJobs="1"
Priority="1" ScenarioName="clock450.tick"
SWmodelfilename="icm">
  <ExecutionGraph EGId="clock450.tick"
EGname="clock450.tick" IsMainEG="true"
StartNode="S_clock450.tick">
    <SplitNode NodeId="S_clock450.tick"
NodeName="S_clock450.tick">
      <ExpandedNode
NodeId="X_trajectoryPlanner.go"
NodeName="X_trajectoryPlanner.go"
Probability="1.0"
EGId="E_trajectoryPlanner.go"
EGname="E_trajectoryPlanner.go"/>
    </SplitNode>
  </ExecutionGraph>
  <ExecutionGraph
EGId="E_trajectoryPlanner.go"
EGname="E_trajectoryPlanner.go"
IsMainEG="false"
StartNode="N_trajectoryPlanner.go">
    <BasicNode
NodeId="N_trajectoryPlanner.go"
NodeName="N_trajectoryPlanner.go"
Probability="1.0">
      <ResourceRequirement ResourceId="R_CPU"
UnitsOfService="89.66507"/>
    </BasicNode>
    <BasicNode
NodeId="N_positionMonitor.read"
NodeName="N_positionMonitor.read"
Probability="1.0">
      <ResourceRequirement ResourceId="R_CPU"
UnitsOfService="3.0634942"/>
    </BasicNode>
    <BasicNode NodeId="N_repository.access"
NodeName="N_repository.access"
Probability="1.0">
      <ResourceRequirement ResourceId="R_CPU"
UnitsOfService="19.920586"/>
    </BasicNode>
    <Arc FromNode="N_trajectoryPlanner.go"
ToNode="N_positionMonitor.read"/>
    <Arc FromNode="N_positionMonitor.read"
ToNode="N_repository.access"/>
  </ExecutionGraph>
</PerformanceScenario>
```

Figure 8. S-PMIF for clock450.tick Simple Model

The simple model consists of four performance scenarios. Figure 8 shows the generated S-PMIF for one of them.

The advanced system model has nine scenarios. Figure 9 shows the S-PMIF for the same scenario in the advanced model.

Figure 10 shows the imported models. On the left is a portion of the simple model corresponding to the execution graph for the expanded node, E_trajectoryPlanner.go. Its "no contention" solution is shown. On the right is the generated advanced model consisting of the N_trajectoryPlanner.go basic node followed by two synchronous call nodes.

In order to have a baseline for comparing the results, the controller was analyzed using the worst-case latency prediction capability provided by the PSK performance-reasoning framework. This analysis first transforms the design specification into a performance model in which the response to each external event is expressed as a linear sequence of actions, even if the original response presents branching and internal concurrency. The resulting performance model is then analyzed using the technique for varying priorities in Rate Monotonic Analysis

```
<PerformanceScenario
EGId="trajectoryPlanner.go"
InterarrivalTime="450.0" Priority="4"
ScenarioName="trajectoryPlanner.go"
SWmodelfilename="icm">
  <ExecutionGraph EGId="trajectoryPlanner.go"
EGname="trajectoryPlanner.go" IsMainEG="true"
StartNode="N_trajectoryPlanner.go">
    <BasicNode
NodeId="N_trajectoryPlanner.go"
NodeName="N_trajectoryPlanner.go">
      <ResourceRequirement ResourceId="R_CPU"
UnitsOfService="89.66507"/>
    </BasicNode>
    <SynchronizationNode
NodeId="N_trajectoryPlanner.read"
NodeName="N_trajectoryPlanner.read"
myType="SynchronousCall"
partnerID="N_positionMonitor.read"
partnerPerfScenarioName="positionMonitor.read
"/>
    <SynchronizationNode
NodeId="N_trajectoryPlanner.put"
NodeName="N_trajectoryPlanner.put"
myType="SynchronousCall"
partnerID="N_repository.access"
partnerPerfScenarioName="repository.access"/>
    <Arc FromNode="N_trajectoryPlanner.go"
ToNode="N_trajectoryPlanner.read"/>
    <Arc FromNode="N_trajectoryPlanner.read"
ToNode="N_trajectoryPlanner.put"/>
  </ExecutionGraph>
</PerformanceScenario>
```

Figure 9. S-PMIF for clock450.tick Advanced Model

(RMA) [46]. This analysis is carried out by MAST [12], a third-party tool integrated with the PSK's performance reasoning framework. For each response being analyzed, RMA creates the worst phasing of tasks in order to compute an upper bound for the worst-case latency or response time. Therefore, it is expected that results obtained by other means be no higher than those provided by RMA.

Table 1 shows the performance results. The first two sections are the results from the RMA analysis and a discrete event simulation integrated in the PSK. The third section shows the *SPE·ED* results. The best case is the analytic solution of the *SPE·ED* simple model. The average and worst cases are the simulation solution of the *SPE·ED* advanced system model. As expected, the analytic best case for both RMA and *SPE·ED* are exact. The simulation

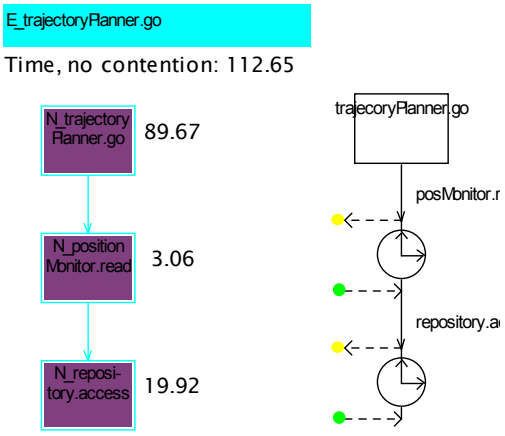

Figure 10. Imported Clock450.tick Simple and Advanced Model

Table 1. Robot Controller Results

Transaction	Best	Average	Worst
RMA Analytic			
clock130.tick	15.04		98.04
clock450.tick	112.65		262.77
clock150.tick	60.02		79.94
clock2000.tick	0.32		278.14
DE Simulation			
clock130.tick	15.04	33.71	75.08
clock450.tick	247.73	259.49	262.83
clock150.tick	60.02	60.00	60.04
clock2000.tick	0.32	103.08	278.20
SPE·ED Results			
clock130.tick	15.04	33.78	99.07
clock450.tick	112.65	259.67	262.77
clock150.tick	60.02	60.02	60.02
clock2000.tick	0.32	71.61	278.14

solutions are also comparable, but not exact. This is especially noticeable in the best case because the discrete event simulation best case does include contention. For example, even in the best case, the response to clock450.tick will be preempted twice by clock150.tick, resulting in a response time higher than the no-contention best case.

The next step is to evaluate an alternative architecture that replaces the X and Y controllers with controllers that also provide position feedback to the position monitor. This changes the scenario for clock150.tick in the simple model to make two additional calls. It changes the ControllerX and ControllerY threads in the advanced model to make asynchronous calls to the PositionMonitor.input. Table 2 shows the results for this architectural alternative.

As before, the best case analytic results are exact. However, these results show some differences in the simulation solutions for the

Table 2. Results for Architectural Alternative

Transaction	Best	Average	Worst
RMA Analytic			
clock130.tick	15.04		124.06
clock450.tick	112.65		496.91
clock150.tick	86.03		109.02
clock2000.tick	0.32		431.24
DE Simulation			
clock130.tick	15.04	52.18	115.99
clock450.tick	314.80	347.63	431.04
clock150.tick	86.03	89.57	105.99
clock2000.tick	16.19	220.18	431.36
SPE·ED Results			
clock130.tick	15.04	46.51	208.16
clock450.tick	112.65	305.60	317.88
clock150.tick	86.03	90.08	192.65
clock2000.tick	0.32	128.68	413.30

advanced model. In particular, *SPE·ED* models have higher worst case times for the clock130.tick and clock150.tick scenarios than RMA analytic results, which should never happen. This is because *SPE·ED* computes the average time for all calls to the positionMonitor.input thread. RMA, however, distinguishes between the calls from the different clocks. For example, positionMonitor.input participates in the responses to clock130 and clock150. The problem is that it will have different response times for each of the clocks. For instance, when participating in clock130, positionMonitor.input could be preempted by an arrival from clock150. That preemption would last for approximately 65ms. However, when participating in clock150, positionMonitor.input obviously would never be preempted by an arrival from clock150. It is possible to compute more precise results manually from *SPE·ED* output.

This proof of concept demonstrates the viability of the model interchange approach for the performance assessment of real-time system architectures. It is helpful to compare the solutions from different software performance modeling tools.

8. CONCLUSIONS

This paper has illustrated the use of a model interchange format to support the performance analysis of real-time systems. It builds on previous work in the areas of component-based systems, software performance engineering, and model interchange. Transformations between the Construction and Composition Language and the Software Performance Model Interchange Format (S-PMIF) were defined for both simple and advanced models. A case study illustrates the process and compares model solutions obtained using the *SPE·ED* software performance engineering tool with those obtained using rate-monotonic analysis and discrete event simulation.

In defining the model transformation, we identified changes to the S-PMIF that were needed for analyzing a real-time design. We also found that preserving the type hierarchy of the S-PMIF meta-model in the schema would facilitate the implementation of S-PMIF interchange support by tools using strongly typed modeling technologies to generate the XML such as EMF or some model transformation languages.

This work has opened a door to allow the performance analysis of CCL specifications with other analysis tools without the need for additional integration effort. This means that standard SPE models can easily be used for analysis of systems specified in CCL.

Finally, this paper has demonstrated the ease with which the S-PMIF can be employed to transform additional design notations (other than UML) into software performance models.

9. REFERENCES

1. Woodside, C.M., G. Franks, and D.C. Petriu. *The Future of Software Performance Engineering*. in *International Conference on Software Engineering (ICSE)*. 2007. Washington, DC: IEEE Computer Society.
2. Hissam, S.A., et al., *Enabling Predictable Assembly*. Journal of Systems and Software: Special Issue on Component-Based Software Engineering, 2003. **65**(3): p. 185-198.
3. Larsson, M., *Predicting quality attributes in component-based software systems*. 2004, Mälardalen University.
4. Liu, V., I. Gorton, and A. Fekete, *Design-level performance prediction of component-based applications*. IEEE Trans. on Software Engineering, 2005. **31**(11): p. 928-941.
5. Merson, P. and S.A. Hissam. *Predictability by construction*. in *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05)*. 2005. San Diego, CA: ACM.
6. Hissam, S.A., G.A. Moreno, and K.C. Wallnau, *Using containers to enforce smart constraints for performance in industrial systems*. 2005, Software Engineering Institute - Carnegie Mellon University: Pittsburgh, PA.
7. Moreno, G.A. *Creating custom containers with generative techniques*. in *Proc. 5th International Conference on Generative Programming and Component Engineering (GPCE06)*. 2006. Portland, OR.
8. Bass, L., et al., *Reasoning Frameworks*. 2005, Software Engineering Institute - Carnegie Mellon University: Pittsburgh, PA.
9. Ivers, J. and G.A. Moreno. *Model-driven development with predictable quality*. in *SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA07)*. 2007. Montreal, Quebec, Canada: ACM.
10. Wallnau, K.C. and J. Ivers, *Snapshot of CCL: A language for predictable assembly*. 2003, Software Engineering Institute - Carnegie Mellon University: Pittsburgh, PA.
11. Hissam, S.A., et al., *Pin component technology (V1.0) and its C interface*. 2005, Software Engineering Institute - Carnegie Mellon University: Pittsburgh, PA.
12. Gonzalez Harbour, M., et al. *MAST: Modeling and Analysis Suite for Real-Time Applications*. in *Proceedings 13th Euromicro Conference on Real-Time Systems (ECRTS)*. 2001. Washington, DC: IEEE Computer Society.
13. Smith, C.U. and C.M. Lladó. *Performance Model Interchange Format (PMIF 2.0): XML Definition and Implementation*. in *Proc. 1st Int. Conf. on Quantitative Evaluation of Systems (QEST)*. 2004. Enschede, NL: IEEE Computer Society.
14. Smith, C.U., et al. *From UML Models to Software Performance Results: An SPE Process Based on XML Interchange Formats*. in *Workshop on Software and Performance (WOSP05)*. 2005. Palma de Mallorca: ACM.
15. Williams, L.G. and C.U. Smith. *Information Requirements for Software Performance Engineering*. in *Proceedings 1995 International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*. 1995. Heidelberg, Germany: Springer.
16. Smith, C.U. and L.G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. 2002, Boston: Addison-Wesley.
17. L&S, *Computer Technology, Inc., Performance Engineering Services Division*, in *# 110, PO Box 9802*, (505) 988-3811, www.spe-ed.com: Austin, TX 78766.
18. Smith, C.U., et al. *Interchange formats for performance models: Experimentation and output*. in *Proc. Quantative Evaluation of Systems (QEST)*. 2007. Edinburgh, Scotland: IEEE.

19. W3C, *World Wide Web Consortium*. 2001, www.w3c.org.
20. Kazman, R., et al., *Scenario-Based Analysis of Software Architecture.* IEEE Software, 1996. **13**(6): p. 47-55.
21. Kazman, R., et al. *The Architecture Tradeoff Analysis Method*. in *International Conference on Engineering of Complex Computer Systems (ICECCS98)*. 1998.
22. Williams, L.G. and C.U. Smith. *PASA^{SM}: A Method for the Performance Assessment of Software Architectures*. in *Proc. 3rd Int. Workshop on Software and Performance*. 2002. Rome, IT: ACM Press.
23. Balsamo, S., P. Inverardi, and C. Mangano. *An Approach to Performance Evaluation of Software Architectures*. in *Workshop on Software and Performance*. 1998. Santa Fe, NM: ACM.
24. Balsamo, S. and M. Marzolla. *Performance Evaluation of UML Software Architectures with Multiclass Queueing Network Models*. in *WOSP 2005*. 2005. Palma de Mallorca: ACM.
25. Gu, G. and D. Petriu. *From UML to LQN by XML Algebra-based Model Transformations*. in *WOSP 2005*. 2005. Palma de Mallorca: ACM.
26. López-Grao, J.P., J. Merseguer, and J. Campos. *From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering*. in *Proc. Workshop on Software and Performance*. 2004. Redwood Shores, CA: ACM.
27. Petriu, D. and C.M. Woodside. *Analyzing Software Performance Requirements Specification for Performance*. in *Proc. Workshop on Software and Performance 2002*. 2002. Rome: ACM.
28. Savino, N., et al. *Extending UML to Manage Performance Models for Software Architectures: A Queuing Network Approach*. in *Proc. 9th Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, SPECTS*. 2002. San Diego, CA.
29. Smith, C.U. and C.M. Lladó, *Performance Model Interchange Format (PMIF 2.0): XML Definition and Implementation Update Technical Report.*. 2007, L&S Computer Technology, Inc.: Santa Fe, NM.
30. Smith, C.U., et al. *From UML models to software performance results: An SPE process based on XML interchange formats*. in *Proc. 5th Int. Workshop on Software and Performance*. 2005. Palma, Illes Balears, Spain: ACM Press.
31. Woodside, C.M., et al. *Performance by Unified Model Analysis (PUMA)*. in *WOSP 2005*. 2005. Palma de Mallorca: ACM.
32. D'Ambrogio, A. *A Model Transformation Framework for the Automated Building of Performance Models from UML Models*. in *Proc. 2005 Workshop on Software and Performance*. 2005. Palma de Mallorca: ACM Press.
33. Bertolino, A., et al., *From UML to Execution Graphs and Queueing Networks: Design and Implementation of the XML-based tool XPRIMAT*. 2004, Universita del L'Aquila: L'Aquila, Italy.
34. Cortellessa, V., M. Gentile, and M. Pizzuti. *XPRIT: An XML-based Tool to Translate UML Diagrams into Execution Graphs and Queueing Networks (Tool Paper)*. in *Proc. of 1st Int. Conf. on the Quantitative Evaluation of Systems*. 2004. Enschede, NL: IEEE Computer Society.
35. Gomaa, H. and D.A. Menasce, *Performance Engineering of Component-Based Distributed Software Systems*, in *LNCS 2047: Performance Engineering State of the Art and Current Trends*, Dumke, et al., Editors. 2001, Springer-Verlag: Berlin. p. 40-55.
36. Gu, G. and D. Petriu. *XSLT Transformation from UML Models to LQN Performance Models*. in *Proc. Workshop on Software and Performance*. 2002. Rome: ACM.
37. Wu, X. and C.M. Woodside. *Performance Modeling from Software Components*. in *Proc. Workshop on Software and Performance*. 2004. Redwood Shores, CA: ACM.
38. Becker, S., H. Koziolek, and R. Reussner. *Model-based performance prediction with the Palladio component model*. in *Workshop on Software and Performance (WOSP07)*. 2007. Buenos Aires, Argentina: ACM.
39. Grassi, V., R. Mirandola, and A. Sabetta. *A model-driven approach to performability analysis of dynamically reconfigurable component-based systems*. in *Woskshop on Software and Performance (WOSP07)*. 2007. Buenos Aires, Argentina: ACM.
40. Cortellessa, V. *How Far Are We From the Definition of a Common Software Performance Ontology?* in *WOSP 2005*. 2005. Palma de Mallorca: ACM.
41. Budinsky, F., E. Merks, and D. Steinberg, *Eclipse Modeling Framework 2.0 (2nd Edition)*. 2006: Addison-Wesley Professional.
42. Smith, C.U. and L.G. Williams, *Performance Engineering Evaluation of CORBA-based Distributed Systems with SPEED*, in *Lecture Notes in Computer Science*, R. Puigjaner, Editor. 1998, Springer: Berlin, Germany.
43. Smith, C.U. and L.G. Williams, *Performance Engineering of Object-Oriented Systems with SPEED*, in *Lecture Notes in Computer Science 1245: Computer Performance Evaluation*, M. R., et al., Editors. 1997, Springer: Berlin, Germany. p. 135-154.
44. Hughes, P., *SP Principles*. 1988, STC Technology.
45. Beilner, H., J. Mäter, and N. Weissenburg. *Towards a Performance Modeling Environment: News on HIT*. in *Proceedings 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*. 1988: Plenum Publishing.
46. Gonzalez Harbour, M., M. Klein, and J. Lehoczky, *Timing analysis for fixed-priority scheduling of hard real-time systems*. IEEE Trans. on Software Engineering, 1994. **20**(1): p. 13-28.