Performance Evaluation ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Performance analysis of real-time component architectures: An enhanced model interchange approach

Gabriel A. Moreno [a,1], Connie U. Smith [b,*]

[a] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA
[b] Performance Engineering Services, PO Box 2640, Santa Fe, NM 87504, USA

ARTICLE INFO

Article history:
Received 20 January 2009
Received in revised form 24 July 2009
Accepted 30 July 2009
Available online xxxx

Keywords:
Software performance engineering
Performance model interchange
Model driven architecture analysis
Component based real-time systems
Model to model transformation

ABSTRACT

Model interchange approaches support the analysis of software architecture and design by enabling a variety of tools to exchange performance models using a common schema. This paper builds on the Software Performance Model Interchange Format (S-PMIF), extending it to support the analysis of real-time systems and adapting it to be suitable for implementation with modeling frameworks such as MOF or EMF. This enhances the model interchange process by making it possible to define model-to-model transformations from design models into software performance models. The paper addresses real-time system designs expressed in CCL and their transformation into the S-PMIF for additional performance analyses.

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.

An earlier version of this paper illustrated an approach to making performance analysis more accessible [2]. It made several contributions:

* Corresponding author. Tel.: +1 505 988 3811.
E-mail addresses: gmoreno@sei.cmu.edu (G.A. Moreno), cusmith@spe-ed.com (C.U. Smith).
URL: http://www.spe-ed.com (C.U. Smith).
1 Tel.: +1 412 268 1213.

0166-5316/$ – see front matter © 2009 Published by Elsevier B.V.
doi:10.1016/j.peva.2009.07.008

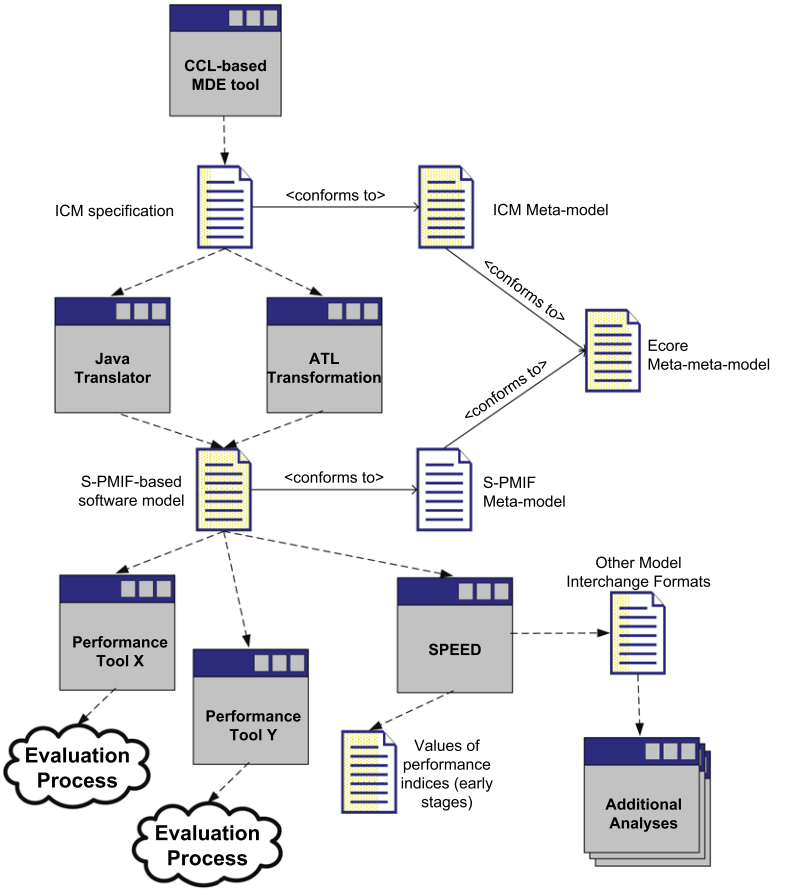

Fig. 1. Overview of tools, models and meta-models.

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

This paper enhances the earlier work with:

- An extended S-PMIF 2.0 meta-model, suitable for implementation with Ecore [3], that enables an automated model-to-model transformation (M2M) from design models to performance models.
- An additional implementation using the Atlas Transformation Language (ATL) [4] to transform an intermediate constructive model (ICM) to S-PMIF.
- An updated proof of concept to support the new S-PMIF 2.0.
- Additional related work including a discussion of the modeling power of S-PMIF compared to MARTE.
- Conclusions about the viability of M2M for design and performance 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 2.0 meta-model for real-time systems and the adaptations to be definable in Ecore. 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. Fig. 1 provides an overview of relationships between the tools, models and meta-models described in the paper.

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.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

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 [5–7]. The PACC initiative at the Software Engineering Institute proposes the use of smart constraints to achieve predictability by construction [8]. 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 [9,10].

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 [11]. 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) [12]. The PSK is a development environment that includes the Construction and Composition Language (CCL) [13], 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 [14] 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 [15], 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 [16]. This paper describes the use of the Software Performance Model Interchange Format (S-PMIF) [17,18] to allow the analysis of real-time designs specified in CCL with additional performance analysis tools. Section 3.2 discusses the factors that led to our selection of S-PMIF as the model interchange format.

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 [19].

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 calls for software performance models that specify software processing steps (basic, case, repetition, synchronization, etc.), software resource requirements for each, and computer resource requirements for each software resource. Software models are mapped onto system performance models that represent computer resources and (computed) demands for software scenarios. An advanced model evaluates software communication and synchronization and its impact on system

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

performance. Software models are graph models; system models are queueing network models (QNM); and advanced models may be either simulation models or approximated with layered queueing networks (LQN).

SPE· ED [20] is a tool designed specifically to support the SPE methods and models defined in [19]. 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.

No contention solves the software model only. The Contention solution is an analytic solution of a portion of the system model whereas the System model solution is a simulation solution of the complete system model.

The simple model solution (no contention) suffices for most performance analyses early in development. The data that is available at that time usually does not 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*, the user can automatically create an advanced system execution model when s/he needs to quantify synchronization 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) [17,18] was the first common representation for exchanging 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 trade-offs 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 [21] 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. Additional tools would require a custom interface to every other tool, resulting in a $N \cdot (N - 1)$ requirement for customized

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

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 [22]. 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 using XML as the interface is insignificant compared to a fine-grained interface that exchanges each XML element as it is generated.

Importing and exporting models fits very well within the context of Model Driven Engineering (MDE) [23,24], an approach to software development based on models and transformations between them. Model Driven Architecture (MDA) is the Object Management Group's (OMG) approach to MDE. MDA enables the development of a Platform-Independent Model (PIM) of the application's business functionality and behavior, transforms it through one or more Platform-Specific Models (PSMs), and produces generated code and a deployable application. The PIM remains stable as technology evolves, thus portability and interoperability are built into the architecture. These models have to conform to meta-models. The Meta-Object Facility (MOF) [25] is a standard meta-meta-model that allows defining meta-models in terms of object-oriented concepts such as classes, inheritance and associations. Although MDA focuses on the generation of software implementations from models, the same technologies can be used for other purposes, such as transforming a design model into a performance model for analysis.

Model transformation tools based on MOF or other modeling frameworks such as the Eclipse Modeling Framework (EMF) [3] may be used to automate M2M transformations, thus eliminating the need for custom coding of many of the import and export functions for tools that support a MOF compliant interface. This further reduces the requirement for customized interfaces from $2 \cdot N$ per tool to $2 \cdot N$ per meta-model when M2M transformations are viable.

Simple M2M transformations are not always viable. For example, transformation from S-PMIF to PMIF requires an algorithmic solution of a software execution model to derive the model parameters for a system execution model that would be represented by PMIF. The reverse transformation, however, is structural in nature and does not require solving the system execution model, so a M2M transformation is viable.

The Modeling and Analysis of Real-Time Embedded systems (MARTE) is OMG's UML profile that supports specification, design, verification/validation, and analysis of Real-time and Embedded Systems (RTES) [26]. It is a replacement for the UML Profile for Schedulability, Performance and Time (SPT). MARTE defines core concepts for a model-based description of an RTES design that support existing performance and schedulability analysis techniques with a tool interoperability approach. It has a rich set of elements for specifying RTES behavior including:

- Clock management and timing requirements for start and stop conditions, miss ratios, maximum jitter, etc.
- Resource specifications for storage and energy consumption.
- Communication constraints such as shared data, messages, queue sizes, and communication overhead.
- Mutual exclusion and event notification, conditions for activation and termination.
- Additional types of distributions of random variables including Bernoulli, Binomial, Gamma, Geometric, and Histogram.

Some of these elements can be mapped onto S-PMIF synchronization nodes, such as mutual exclusion, event notification, and communication constraints. Communication overhead can be handled with S-PMIF computer resource requirements. Many other elements exceed the analysis capabilities of the queueing network paradigm covered by S-PMIF, such as energy consumption, storage requirements, and clock management. So, S-PMIF is not a complete solution for the analysis of MARTE-specified RTES, but it does support many important performance and schedulability architectural assessments.

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) [27] 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) [28] 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, trade-offs between quality requirements can be evaluated.

PASA^{SM} [29] is a method for the performance assessment of software architectures. It uses the principles and techniques of SPE [19] to identify potential areas of risk within the architecture with respect to performance and quality requirements. 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 [19]. 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 trade-offs may be required. While PASA's primary focus is on performance, quality attributes and trade-offs between them are considered as well.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

Earlier approaches to architecture assessment (e.g., [30–34], and [35]) 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, [16,36] 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 first defined [18] and later extended [37]. 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. later developed the Core Scenario Model (CSM) that combines software and system models based on LQN in [38]. D'Ambrogio also defines a MOF meta-model of LQNs and transfers UML models to LQNs in [39]. Grassi et al. subsequently developed a meta-model, KLAPER [40,41], that is similar to CSM in its LQN approach, but has a different point of view on the important primitives (and supports reliability analysis using SHARPE). All of these meta-models are tailored to LQN (the advanced model in the SPE approach). So, for example, they are best suited to determining if performance requirements are met, analyzing capacity requirements, and changing the assignments of components to computer resources. If performance problems are found, they give limited insight into how component(s) could be redesigned to meet requirements. They go directly from a software design to an advanced model, so the ability to discover and correct problems with simpler graph-based models is not supported. For example, if performance requirements for an RTES design cannot be met with one user, there is no need to solve a contention model with hundreds of users. Furthermore, the software design model provides insight into the *design* problems, whereas the LQN model primarily provides system-level feedback. This is the primary reason our work uses S-PMIF for the model interchange format. Note that it is possible to transform S-PMIF into CSM or KLAPER so we do not sacrifice modeling power with this choice.

Other approaches have focused on transferring information between UML-based software design tools and software performance engineering tools, such as [42–44,17]. Gu and Petriu [45] and Balsamo and Marzolla [31] 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.

Recent work has investigated M2M approaches for performance model interoperability. Cortellessa et al. examine ATL as an alternative to custom Java code for transforming UML software models into PMIF queueing network models [46]. One of the difficulties they note is traceability and reversibility from the queueing network model back to the design model, to uniquely and automatically identify the critical element in the software model that caused performance problems. Our work uses the S-PMIF software performance model interchange format, which explicitly represents the software processing steps so the critical elements can be easily identified. Becker examines M2M transformations from PIM design models into PSM realizations of the design, and incorporates performance-determining implementation choices into performance models by using Coupled Transformations [47]. His results show improved precision in performance predictions (over methods that do not specify resource requirements of implementation choices) with the use of his M2M approach. Gherbi and Khendek use ATL to transform an Ecore subset of UML/SPT models (representing only the SAProfile Package) into an analysis model conforming to their specific Ecore Schedulability Analysis Meta-model [48]. They illustrate the transformation with a simple real-time system proof of concept. Their work establishes the viability of using M2M for transforming design models into analysis models; but the result is a customized transformation to one specific analysis tool rather than a common interchange format that would work with multiple tools. They also note that the analysis result is whether or not the tasks are schedulable but, if not, there is no indication of the software design problem, because of the lack of connection of quantitative results back to the design model.

This body of work demonstrates that model interoperability among a set of tools is viable. Common interchange formats such as PMIF, S-PMIF, and CSM 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 [49]. 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 [50]. 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 [51]. 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

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

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 [52]. 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) [13]. 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 [13]. 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.

```
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[2] 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 [13].

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

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

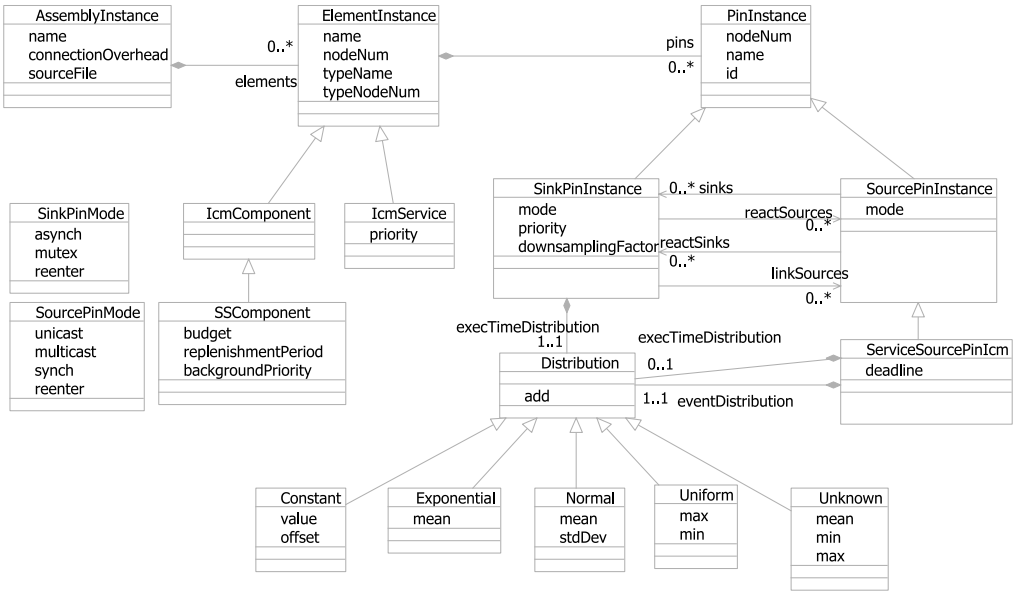

Fig. 2. ICM meta-model.

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 Fig. 2, 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 [19]. The SPE meta-model class diagram is shown in Fig. 3. The complete definition is available at www.spe-ed.com/pmif/

An earlier version of this work [2] changed the original meta-model described in [37] to support the specification and analysis of real-time systems. This paper presents a substantially modified S-PMIF, version 2.0, that adapts the meta-model so that it can be expressed in terms of the Ecore meta-meta-model—the core meta-model of the Eclipse Modeling Framework—and restructures the way some key elements are represented in order to align it with the PMIF, LQN, and ICM meta-models to facilitate M2M transformations.

The original real-time extensions included 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*. *PerformanceScenarios* 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*.

The primary change to support Ecore is to convert the associative entities in the previous meta-model, *OverheadMatrix* and *Arc*, into either classes or attributes in the new meta-model. The *Arc* became a class with attributes fromNode and

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

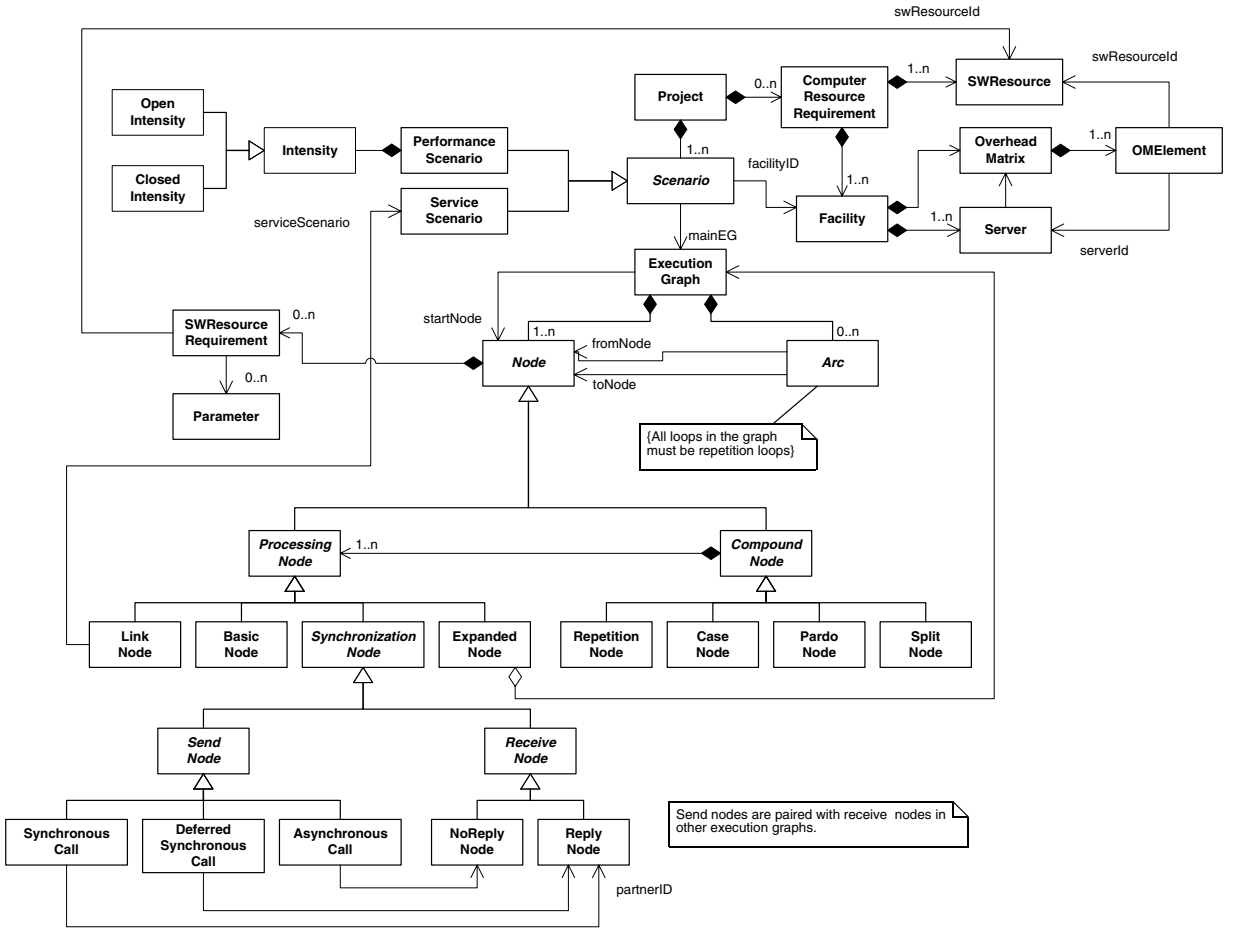

Fig. 3. S-PMIF 2.0 meta-model.

toNode. This facilitates adding other attributes to *Arcs* in the future such as graphical coordinates or distinguishing types of control flow, such as local and remote procedure calls, as in [53].

To convert the *OverheadMatrix*, we restructured the entire section of the meta-model which represents the relationship of software resources, devices, and the overhead matrix. The upper-right section of Fig. 3 shows the revisions. Now a *Project* consists of one or more *Scenarios* and zero or more *ComputerResourceRequirements*[3]. A *ComputerResourceRequirement* consists of one or more *SWResources* and *Facilities*. Each *Scenario* executes on a *Facility*, and each *Facility* has an *OverheadMatrix*. A *Facility* consists of one or more *Servers*. The term *Device* in the previous meta-model was changed to *Server* to align with the PMIF and because a software performance model may use servers such as a network or a delay that are not "devices". An *OverheadMatrix* consists of one or more *OMElements* that specify the amount of computer processing required on one or more *Servers* for each *SWResource*. Finally, a *Node* in an *ExecutionGraph* specifies zero or more *SWResourceRequirements* that specify an amount of service required from the *SWResources*. *SWResourceRequirements* may be specified using *Parameters* which may be assigned values that will be propagated to all *Nodes* that use them. All *Nodes* do not have *SWResourceRequirements*; for example, *LinkNode* and *ExpandedNode* resource requirements are embedded in the associated *ExecutionGraphs*.

The overhead matrix is usually sparse, so the S-PMIF exchanges it by including the FacilityID and the SWResourceID attributes with each OMElement value. Tools such as *SPE· ED* may convert the model information into a matrix for efficient retrieval of the overhead values.

Earlier extensions to support real-time concepts and align with the ICM added attributes to the meta-model to allow specification of the following real-time concepts:

- arrivalDistribution (Scenario) and serviceDistribution (Server). The supported distributions are exponential, normal, constant, erlang, hyperexponential, and uniform(u1,u2). The distribution specification is optional; the default is exponential.

[3] We allow zero *ComputerResourceRequirements* because they may be inserted after a design model is transformed, but the software model cannot be solved without them.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

- schedulingPolicy (Facility). This attribute is an enumerated type (FCFS, IS, LCFSPR, PR, PS, RR, RM) and is optional.
- responseTimeRequirement and throughputRequirement (Scenario). The values of these attributes are real numbers.

In addition, the attributes partnerNodeID and partnerScenario (IDREF) were added to SendNode and attributes were removed from SynchronizationNode.

Several other minor changes were made to better align S-PMIF 2.0 with other performance-related meta-models, to clarify terminology, and simplify the M2M transformations:

- The arrival and service distributions were converted from attributes into a distinct class for each type of distribution that inherit common properties from a *Distribution* class. The structure matches that in the ICM meta-model in Fig. 2. An *Unknown* distribution type was added to S-PMIF to match the ICM.
- A new Deadline attribute was added to *ExecutionGraphs* to match the ICM. Specifying deadlines for execution graphs allows one to have a deadline for an overall scenario, as well as for subsets of processing steps. The ResponseTimeRequirement for a *Scenario* can be used to specify the overall deadline.
- The relationship from a *CompoundNode* to a *ProcessingNode* was changed from aggregation to containment. If there is a CompoundNode C1 that contains a ProcessingNode N1, Node N1 cannot separately exist in the *ExecutionGraph*- it can only exist in C1. There can be a *ProcessingNode* N2 that occurs in the *ExecutionGraph* without being contained in a *CompoundNode*, but we allow an *ExecutionGraph* to contain 1..n *Nodes* that may be *ProcessingNodes*.
- *CompoundNodes* no longer contain *Arcs* for attached nodes because the internal arcs from the *CompoundNode* to the attached *ProcessingNodes* are implicit and have no other function in an *ExecutionGraph*.
- A new attribute ServerKind with an enumerated type (Server or WorkUnitServer) was added, and the attribute name DeviceFeature was changed to ServerRequest (type) to better match the PMIF.
- *SWResourceRequirements* were moved to the *Node* element (rather than *BasicNode* and *CompoundNode*) to allow specifications for additional types of *Nodes*.
- An *ExecutionGraph* now specifies its startNode, but does not have isMainEG because the *Scenario* points to the main execution graph with attribute MainEG.
- Attributes in *LinkNode* were changed because they do not necessarily correspond to *PerformanceScenarios*, they may be *ServiceScenarios*.
- Some attribute names for IDs and IDREFs were changed to better document the connections, e.g., PS.EGId to MainEG.

The S-PMIF is implemented using three separate schemas: Topology, ComputerResourceRequirements, and Distribution[4]. Topology may include ComputerResourceRequirements, and both may include Distribution. This is useful because one may use one of the schemas without using the other. For example, if the computer resource requirements specifications come from another source, it does not need to be included in the topology, and vice-versa.

This extended version of S-PMIF is substantially different from the 2005 version. So, for instance, prototypes developed for the earlier version would have to be modified if they are to support these additional features. Model interchange formats and interfaces, however, must be relatively stable for the model interoperability approach to be viable. S-PMIF was based on concepts embodied in two earlier model interchange formats: the Electronic Data Interchange Format (EDIF) for VLSI designs [54] and the Case Data Interchange Format (CDIF) for software design interchange (also based on EDIF) [55]. Creators of EDIF envisioned the stability problem and addressed it by (1) Using a concept of *levels* that add functionality at each successive level and (2) Giving ownership of EDIF to a standards organization that managed changes.

This extended version of S-PMIF adds a *level* of functionality that includes features for analyzing Real-time systems. We envision other levels to add features for additional types of analysis. Tools can continue to support a lower level without change, or may opt to modify interfaces to support additional functionality and/or other changes. A few of these changes provide a better basis for adding functionality, but it is not essential that they be supported in earlier prototypes. The changes for the Overhead Matrix, however, will be retrofitted into the 2005 version because they are a major restructuring of the schema. Earlier prototypes do not need to be updated, but future work should use the newer version, even for the basic level. Using a standards organization to manage the contents of this and other interchange formats should be considered in the future.

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 [3]. EMF can generate the Java implementation classes to load, manipulate and persist instances of the model. The S-PMIF XML schema

[4] The previous OverheadMatrix schema was renamed ComputerResourceRequirement to match the terminology used in [53,19]. The Device schema was included in it. The Distribution schema is new.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

```
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 SWResourceRequirement to node from sink.execTimeDistribution
  lastNode = node
  for each source in sink.reactSources {
    newNode = visitSource(source)
    arc = new Arc
    arc.from = lastNode
    arc.to = newNode
    lastNode = newNode
  }
  return node
}
```

Fig. 4. Pseudocode for simple S-PMIF model generation.

was converted to an Ecore meta-model using EMF. This allowed us to generate the Java implementation to manipulate the S-PMIF models with EMF and also to use the S-PMIF Ecore meta-model as a target of the model transformation from ICM to S-PMIF.

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. In Sections 6.1 and 6.2 we present the pseudo-code for transformations that were implemented in Java, using the classes generated by EMF. In Section 6.4, we present the generation of an advanced model using a model transformation language.

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. When visiting an asynchronous source pin, a *SplitNode* is created in the target S-PMIF model to represent the initiation of concurrent threads of execution. When visiting a sink pin, a *BasicNode* is created to represent the computation associated with the pin, and arcs are created to represent the order of execution of the nodes that follow.

Fig. 4 shows the pseudocode for the two functions that implement the core of the transformation described above. 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 *SWResourceRequirement* 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.

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 [17]. For that reason, in the Java implementation generated with EMF from the original S-PMIF schema, *Node*,*BasicNode*, and *SplitNode* had no subtype relationship. This complicated 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 made 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 [17], 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, although 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 similar to the original S-PMIF schema so that

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

```
<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>
```

Fig. 5. Containment with schema choice.

existing tools could easily adopt the schema for S-PMIF 2.0. 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 Fig. 5 was changed, as it appears in Fig. 6. 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 was different from the output compliant with the original S-PMIF schema, XSD substitution groups were defined so that the desired XML output 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'' .../>

The schema was also enhanced to better represent references between instances to simplify the M2M transformation. For example, the XSD schema specifies ID and IDREF attributes that establish a relationship among entities, such as the StartNode attribute of an execution graph that specifies the ID of the start node, rather than a pointer to the specific Node instance desired. This requires extra implementation code to retrieve the desired instance. More importantly, XML validation tools only check that a model is syntactically correct, but a syntactically correct model may not be semantically correct. For example, XML requires that the StartNode be a valid ID, but that ID might be associated with a *Server* rather than a *Node* and the model would still be syntactically valid.

Ecore has the ability to better specify these references, so all IDREFs were supplemented with Ecore references. The following shows the Ecore reference for StartNode:

< xs:attribute name=''StartNode'' type=''xs:IDREF'' use=''required'' ecore:reference=''spmif:Node_type''/>

In this way, references in the model can be easily navigated as it is intended in the meta-model, without the need for searching for an instance by ID. The following statement shows how simple it becomes to obtain the starting node of an execution graph.

```
start = eg.getStartNode();
```

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 overall approach to generate the advanced model is to create a performance scenario for each sink pin in the assembly. Each of these scenarios 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 *SWResourceRequirement* 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.

The pseudocode for the algorithm used to generate the advanced model is shown in Fig. 7. The most important function is *getPSForSink*. This function creates the scenario for a sink pin in the assembly only if it has not created it before; otherwise, it returns the already existing scenario. 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.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

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

Fig. 6. Containment with base type.

```
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 SWResourceRequirement 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.partnerScenario = getPSForSink(linkedSink, false)
      arc = new Arc
      arc.from = lastNode
      arc.to = node
      lastNode = node
    }
  }
  psMap[sink] = ps
  return ps
}
```

Fig. 7. Pseudocode for advanced S-PMIF model generation.

6.4. ATL transformation

In this section, we present a transformation from ICM to an advanced S-PMIF 2.0 model using the ATLAS Transformation Language (ATL) [4]. ATL is a hybrid model transformation language because it supports both declarative and imperative constructs. Since the transformations presented in the previous sections were implemented in Java, and therefore in an imperative style, we implemented this transformation in ATL using the declarative style as much as possible, which is the style encouraged by ATL.

An ATL transformation takes an input model that conforms to a source meta-model and produces an output model that conforms to a target meta-model. The transformation is specified as a set of transformation rules. Matched rules, the kind of rules used in declarative style, specify a source pattern and a target pattern. The source pattern consists of one or more types from the source meta-model, with an optional guard. When the source pattern is matched in the input model, the elements in the target pattern of the rule are created in the output model. Bindings in the target pattern allow the initialization of features (i.e., attributes and associations) of the created elements. Regular matched rules execute when they match elements in the input model. There is a special kind of matched rule, the *lazy rule* that is triggered only when it is referred to by other rules.

Figs. 8–11 show the source code of the ATL transformation. The overall approach in this transformation is the same as that described in Section 6.3. Fig. 8 shows the initial declaration of the module, stating the input and output meta-models. It also defines some helpers that are used in the transformation rules. For instance, the helper *isTopLevel* returns true if a sink pin is directly connected to a service source pin. The rule *Assembly2Project* shown in Fig. 9 is executed for instances of *AssemblyInstance* type of the ICM meta-model found in the input model. The target pattern of the rule indicates that it creates an instance of *ProjectType* of the S-PMIF 2.0 meta-model when it is executed. The bindings in the target pattern,

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

```
--@atlcompiler atl2006
module advanced;
create OUT : SPMIF from IN : ICM;

helper context ICM!ElementInstance def : getSourcePins()
        : Set(ICM!SourcePinInstance) =
    self.pins->select(p | p.oclIsKindOf(ICM!SourcePinInstance));

helper context ICM!ElementInstance def : getSinkPins()
        : Set(ICM!SinkPinInstance) =
    self.pins->select(p | p.oclIsKindOf(ICM!SinkPinInstance));

helper context ICM!PinInstance def : getFullName() : String =
    self.elementInstance.name + '.' + self.name;

-- a top level sink pin is one connected to a service source pin (first in a response)
helper context ICM!SinkPinInstance def : isTopLevel() : Boolean =
    ICM!ServiceSourcePinIcm.allInstances()
                    ->collect(src | src.sinks)->flatten()->includes(self);

helper def : CPU : SPMIF!SoftwareResourceType = 0;

helper context ICM!Constant def : computedMean() : Real = self.value;
helper context ICM!Unknown def : computedMean() : Real = self.mean;
helper context ICM!Uniform def : computedMean() : Real = (self.max + self.min) / 2.0;
helper context ICM!Normal def : computedMean() : Real = self.mean;
helper context ICM!Exponential def : computedMean() : Real = self.mean;
```

Fig. 8. ATL transformation source code (part 1).

```
rule SetupComputerResources() {
    to
        swResource : SPMIF!SoftwareResourceType (
            sWResourceId <- 'R_CPU',
            sWResourceName <- 'CPU'
        ),
        server : SPMIF!ServerType (
            serverId <- 'D_CPU',
            serverName <- 'CPU',
            quantity <- 1,
            serviceUnits <- 'cpuTime',
            serviceTime <- 1.0,
            serverRequest <- #NonFCFSDemandSpec,
            schedulingPolicy <- #PS
        ),
        omElement : SPMIF!OMElementType (
            sWResourceId <- swResource,
            serverId <- server,
            amountOfService <- 1.0
        ),
        overheadMatrix : SPMIF!OverheadMatrixType (
            oMElement <- Sequence { omElement }
        ),
        facility : SPMIF!FacilityType (
            facilityId <- 'comp1',
            facilityName <- 'computer',
            server <- Sequence { server },
            overheadMatrix <- overheadMatrix
        ),
        computerResourceRequirement : SPMIF!ComputerResourceRequirementType (
            facility <- Sequence { facility },
            softwareResource <- Sequence { swResource }
        )
    do {
        thisModule.CPU <- swResource;
        computerResourceRequirement;
    }
}

rule Assembly2Project {
    from
        assembly : ICM!AssemblyInstance
    to
        project : SPMIF!ProjectType (
            projectName <- assembly.name,
            computerResourceRequirement <- thisModule.SetupComputerResources(),

            -- create one performance scenario for each threaded sink pin
            performanceScenario <- assembly.elements
                                    ->collect(s | s.getSinkPins())->flatten()
                                        ->select(e | e.mode <> #reenter)
        )
}
```

Fig. 9. ATL transformation source code (part 2).

introduced by the <— operator, indicate how the features of *project* are initialized. It can be seen that the project name is initialized with the name of the assembly. The *computerResourceRequirement* reference is initialized with the result of a

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

```
abstract rule SinkPin2ScenarioBase {
    from
        sink: ICM!SinkPinInstance
    using {
        name : String = sink.getFullName();
    }
    to
        ps: SPMIF!PSType
        (
            scenarioId <- name,
            scenarioName <- name,
            priority <- sink.priority,
            sWmodelfilename <- 'icm',
            mainEG <- eg,
            executionGraph <- Sequence { eg }
        ),
        eg: SPMIF!EGType (
            eGname <- name,
            eGId <- name,
            node <- Sequence {node,
                        sink.reactSources->collect(s |
                            s.sinks->collect(sink |
                                thisModule.Sink2SynchronizationNode(s, sink)))
                    }->flatten(),
            arc <- eg.node->iterate(n;
                            data : TupleType(arcs : Sequence(SPMIF!ArcType),
                                             lastNode : SPMIF!NodeType)
                                = Tuple{lastNode = OclUndefined} |
                        if data.lastNode = OclUndefined then
                            Tuple{arcs = Sequence{}, lastNode = n}
                        else
                            Tuple{arcs = data.arcs->append(
                                    thisModule.ArcForNodes(data.lastNode, n)),
                                lastNode = n}
                        endif
                    ).arcs,
            startNode <- node
        ),
        swResourceRequirement : SPMIF!SWResourceRequirementType (
            sWResourceId <- thisModule.CPU,
            unitsOfService <- sink.execTimeDistribution.computedMean()
        ),
        node : SPMIF!NodeType (
            sWResourceRequirement <- swResourceRequirement,
            nodeName <- 'N_' + name,
            nodeId <- 'N_' + name
        )
}
```

Fig. 10. ATL transformation source code (part 3).

called rule named *SetupComputerResources* (also shown in Fig. 9). Although called rules are considered part of the imperative style in ATL because they are akin to a procedure call, we have used one here only to avoid the clutter in the *Assembly2Project* rule.

The binding initializing *performanceScenario*, the collection of performance scenarios in the project, is interesting for two reasons. First, it shows how simple it is to navigate the model using OCL expressions in ATL. In this case, it looks for all the elements in the assembly and collects the sink pins for all of them in a flat sequence. Then it selects only those sink pins whose mode is not reentrant (i.e., threaded sink pins). The second interesting thing to note is that the result of the OCL expression in that binding is a sequence of *ICM!SinkPinInstance*, when we are initializing a collection of *SPMIF!PSType*. ATL finds other matched rules to transform these sink pins. In this case, the rule *TopLevelSinkPin2Scenario* (Fig. 11) will be executed. The link between matched source elements and the generated target elements is maintained so that, if the same sink pin is referred to in a binding, it results in the same performance scenario instance.

ATL supports rule inheritance. The rule *SinkPin2ScenarioBase* shown in Fig. 10 is an abstract rule that has all the common declarations needed to create the performance scenario corresponding to a sink pin. In addition to creating the performance scenario instance, the rule also creates the execution graph for the scenario, a node with its corresponding *SWResourceRequirement* representing the execution of the computation of the sink pin, the synchronization nodes to invoke other components if needed, and the arcs connecting the nodes in the execution graph. The rules *SinkPin2Scenario* and *TopLevelSinkPin2Scenario* shown in Fig. 11 extend *SinkPin2ScenarioBase*, adding the bindings that are specific to each case.*SinkPin2Scenario* creates a synchronization node to accept invocations from other scenarios. *TopLevelSinkPin2Scenario* creates a basic node to represent the execution of the sink pin and sets the interarrival interval of the performance scenario to the mean of the interarrival distribution of the service source pin that invokes the sink pin matched by the rule.

The rule *ArcForNodes* (Fig. 11) creates an arc connecting two nodes, and the rule *Sink2SynchronizationNode* (Fig. 11) creates a synchronization node representing the calling end of the connection between source and sink pins. These two rules are used by *SinkPin2ScenarioBase* to create parts of the target elements needed to define the performance scenario.

The ATL transformation from ICM to advanced S-PMIF 2.0 produces the same result as the Java-based transformation described in Section 6.3 with the exception of the order in which performance scenarios appear in the resulting XML file.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

```
rule SinkPin2Scenario extends SinkPin2ScenarioBase {
    from
        sink: ICM!SinkPinInstance (not sink.isTopLevel())
    to
        ps: SPMIF!PSType
        (
            interarrivalTime <- 0.0,
            numberOfJobs <- 1
        ),
        node : SPMIF!SynchronizationNodeType (
            myType <- if sink.mode = #asynch then #NoReply else #Reply endif
        )
}

rule TopLevelSinkPin2Scenario extends SinkPin2ScenarioBase {
    from
        sink: ICM!SinkPinInstance (sink.isTopLevel())
    to
        ps: SPMIF!PSType
        (
            interarrivalTime <- sink.linkSources
                                    ->first().eventDistribution.computedMean()
        ),
        node : SPMIF!BasicNodeType (
            sWResourceRequirement <- swResourceRequirement,
            nodeName <- 'N_' + name,
            nodeId <- 'N_' + name
        )
}

lazy rule ArcForNodes {
    from
        fromNode : SPMIF!NodeType,
        toNode : SPMIF!NodeType
    to
        arc : SPMIF!ArcType (
            fromNode <- fromNode,
            toNode <- toNode
        )
}

lazy rule Sink2SynchronizationNode {
    from
        source : ICM!SourcePinInstance,
        sink : ICM!SinkPinInstance
    using {
        name : String = 'N_' + source.getFullName() + '_' + sink.getFullName();
    }
    to
        synchronizationNode : SPMIF!SynchronizationNodeType (
            nodeName <- name,
            nodeId <- name,
            myType <- if sink.mode = #asynch then
                          #AsynchronousCall
                      else
                          #SynchronousCall
                      endif,
            partnerId <- thisModule.resolveTemp(sink, 'node'),
            partnerScenario <- sink
        )
}
```

Fig. 11. ATL transformation source code (part 4).

For the most part, using ATL to implement the transformation has advantages over implementing it in Java. For example, there is no need to deal with loading and persisting a model because that is done by the ATL environment. In general, the transformation code is more compact in ATL. For example, the selection of sink pins in the binding of the *performanceScenario* feature in the rule *Assembly2Project* takes approximately ten lines of Java code even with the use of the code generated by EMF. In addition, rule inheritance not only makes the transformation code more compact, but also shows clearly what is common and what is specific in the transformation of similar elements. There is one part of the transformation that required more code when implementing it declaratively, namely the generation of arcs to connect nodes. In the Java version, a reference to the last node added to the execution graph is kept in a variable and a new arc is created connecting that last node to the newly created node. In the declarative version in ATL, all nodes in an execution graph are created first, and the arcs to connect them are created afterwards. That required more code mainly due to the fact that there is always one fewer arc than nodes. Other disadvantages of ATL, such as tool usability issues, are a consequence of the lack of maturity of ATL compared to Java. These issues were also noted by Cortellessa et al. in a study comparing Java with ATL for implementing model transformations for software performance engineering [46].

6.5. Importing the models

The S-PMIF is imported into a software performance modeling tool, like *SPE· ED* [56,57], SP [58], or HIT [59] for performance analysis of the software architecture and design, and evaluation of alternatives. The software performance

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

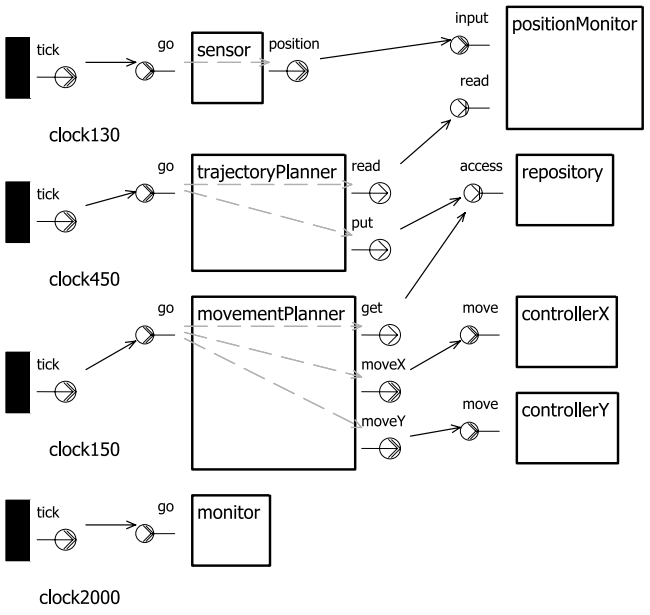

Fig. 12. Robot controller design.

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.

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 [37]. 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 required changes to support the real-time extensions and other features 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.
- *SPE· ED* previously assumed arrival times and service times are exponentially distributed; constant interarrival and service times were implemented for the case study.
- Preemptive-resume scheduling was added.
- An earlier prototype omitted synchronization nodes and the overhead matrix; they were added for this project.

Note that the S-PMIF file looks the same to the importing tool – the importing tool does not need to know whether it was generated with Java code, ATL, or by some other means. So no change to the import function is required.

In this use case, analysis results are reported by the *SPE· ED* tool in its performance visualization format, or in its tables of results. Recent work provides an alternative for automatically producing results in a convenient format for analysts [60]. Future work will consider a reverse transformation that associates output with elements of the design.

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. Fig. 12 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 130 ms). 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.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

```
<PerformanceScenario InterarrivalTime="450.0" MainEG="clock450.tick" NumberOfJobs="1"
                     Priority="1" ScenarioId="clock450.tick"
                     ScenarioName="clock450.tick" SWmodelfilename="icm">
  <ExecutionGraph EGId="clock450.tick" EGname="clock450.tick"
                  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"
                  StartNode="N_trajectoryPlanner.go">
    <BasicNode NodeId="N_trajectoryPlanner.go" NodeName="N_trajectoryPlanner.go"
               Probability="1.0">
      <SWResourceRequirement SWResourceId="R_CPU" UnitsOfService="89.66506958007812"/>
    </BasicNode>
    <BasicNode NodeId="N_positionMonitor.read" NodeName="N_positionMonitor.read"
               Probability="1.0">
      <SWResourceRequirement SWResourceId="R_CPU" UnitsOfService="3.0634942054748535"/>
    </BasicNode>
    <BasicNode NodeId="N_repository.access" NodeName="N_repository.access"
               Probability="1.0">
      <SWResourceRequirement SWResourceId="R_CPU" UnitsOfService="19.92058563232422"/>
    </BasicNode>
    <Arc FromNode="N_trajectoryPlanner.go" ToNode="N_positionMonitor.read"/>
    <Arc FromNode="N_positionMonitor.read" ToNode="N_repository.access"/>
  </ExecutionGraph>
</PerformanceScenario>
```

Fig. 13. S-PMIF for clock450.tick simple 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

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.

The simple model consists of four performance scenarios, one for each source of events. Fig. 13 shows the generated S-PMIF for the scenario corresponding to clock450. The advanced system model has nine scenarios. Fig. 14 shows the S-PMIF for the scenarios involved in the response to clock450 in the advanced model. In this case there are three scenarios that participate in the response, because there are three threads of execution involved in the response. Note that the order of the scenarios in the model interchange file differs in the Java version and the ATL version. This does not affect the imported model or the results.

Fig. 15 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 (RMA) [61]. This analysis is carried out by MAST [15], 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.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

```
<PerformanceScenario InterarrivalTime="450.0" MainEG="trajectoryPlanner.go"
                     Priority="4" ScenarioId="trajectoryPlanner.go"
                     ScenarioName="trajectoryPlanner.go" SWmodelfilename="icm">
  <ExecutionGraph EGId="trajectoryPlanner.go" EGname="trajectoryPlanner.go"
                  StartNode="N_trajectoryPlanner.go">
    <BasicNode NodeId="N_trajectoryPlanner.go" NodeName="N_trajectoryPlanner.go">
      <SWResourceRequirement SWResourceId="R_CPU" UnitsOfService="89.665066"/>
    </BasicNode>
    <SynchronizationNode NodeId="N_trajectoryPlanner.read_positionMonitor.read"
                         NodeName="N_trajectoryPlanner.read_positionMonitor.read"
                         myType="SynchronousCall" partnerId="N_positionMonitor.read"
                         partnerScenario="positionMonitor.read"/>
    <SynchronizationNode NodeId="N_trajectoryPlanner.put_repository.access"
                         NodeName="N_trajectoryPlanner.put_repository.access"
                         myType="SynchronousCall" partnerId="N_repository.access"
                         partnerScenario="repository.access"/>
    <Arc FromNode="N_trajectoryPlanner.go"
         ToNode="N_trajectoryPlanner.read_positionMonitor.read"/>
    <Arc FromNode="N_trajectoryPlanner.read_positionMonitor.read"
         ToNode="N_trajectoryPlanner.put_repository.access"/>
  </ExecutionGraph>
</PerformanceScenario>
<PerformanceScenario InterarrivalTime="0.0" MainEG="repository.access" NumberOfJobs="1"
                     Priority="18" ScenarioId="repository.access"
                     ScenarioName="repository.access" SWmodelfilename="icm">
  <ExecutionGraph EGId="repository.access" EGname="repository.access"
                  StartNode="N_repository.access">
    <SynchronizationNode NodeId="N_repository.access" NodeName="N_repository.access"
                         myType="Reply">
      <SWResourceRequirement SWResourceId="R_CPU" UnitsOfService="19.920586"/>
    </SynchronizationNode>
  </ExecutionGraph>
</PerformanceScenario>
<PerformanceScenario InterarrivalTime="0.0" MainEG="positionMonitor.read"
                     NumberOfJobs="1" Priority="14" ScenarioId="positionMonitor.read"
                     ScenarioName="positionMonitor.read" SWmodelfilename="icm">
  <ExecutionGraph EGId="positionMonitor.read" EGname="positionMonitor.read"
                  StartNode="N_positionMonitor.read">
    <SynchronizationNode NodeId="N_positionMonitor.read"
                         NodeName="N_positionMonitor.read" myType="Reply">
      <SWResourceRequirement SWResourceId="R_CPU" UnitsOfService="3.0634942"/>
    </SynchronizationNode>
  </ExecutionGraph>
</PerformanceScenario>
```

Fig. 14. S-PMIF for clock450.tick advanced model.

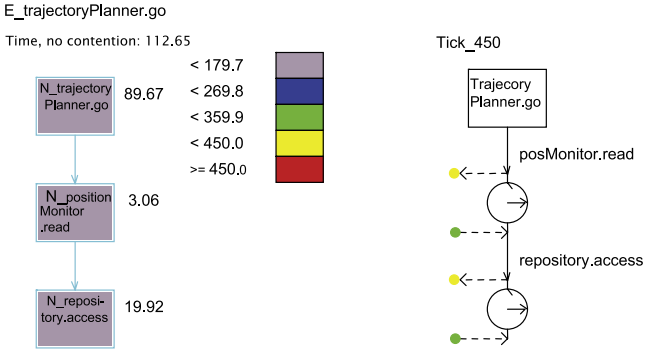

Fig. 15. Imported clock450.tick simple and advanced model.

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 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 best case results are optimistic; however, they provide the ability to identify a failure to meet performance requirements. Those problems must be corrected before more precise analyses are useful.

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.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

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

As before, the best case analytic results are exact. However, these results show some differences in the simulation solutions for the 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 65 ms. However, when participating in clock150, positionMonitor.input obviously would never be preempted by an arrival from clock150. This is not a limitation of S-PMIF. A future implementation of *SPE· ED* will provide results in a format that is more convenient for real-time analysis.

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. Both custom Java transformations and M2M transformations were presented. 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 and associations of the S-PMIF meta-model in the schema facilitates the implementation of S-PMIF interchange support by tools using strongly typed modeling technologies to generate the XML such as EMF or ATL.

The changes in the S-PMIF 2.0 meta-model that make it suitable for implementation with Ecore or MOF allow using model transformation languages to define declaratively the transformation of design models into S-PMIF models for performance evaluation.

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 into software performance models, thus building on the previous UML-based approaches.

Acknowledgment

Lloyd Williams contributed to the earlier, conference version of this paper. He was unable to participate in developing the extensions. Smith's participation was sponsored by US Air Force Contract FA8750 -09-C -0086.

References

[1] C.M. Woodside, G. Franks, D.C. Petriu, The future of software performance engineering, in: International Conference on Software Engineering (ICSE), Washington, DC, May, IEEE Computer Society, 2007, pp. 171–187.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

[2] G.A. Moreno, C.U. Smith, L.G. Williams, Performance analysis of real-time component architectures: A model interchange approach, in: Proc. Workshop on Software and Performance, Princeton, NJ, June, ACM Press, 2008.
[3] F. Budinsky, E. Merks, D. Steinberg, Eclipse Modeling Framework 2.0, 2nd Edition, Addison-Wesley Professional, 2006.
[4] F. Jouault, I. Kurtev, Transforming models with Atl, in: Satellite Events at the MoDELS 2005 Conference, in: Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 128–138.
[5] S.A. Hissam, G.A. Moreno, J.A. Stafford, K.C. Wallnau, Enabling predictable assembly, Journal of Systems and Software: Special Issue on Component-Based Software Engineering 65 (3) (2003) 185–198.
[6] M. Larsson, Predicting Quality Attributes in Component-Based Software Systems, Mälardalen University, 2004.
[7] V. Liu, I. Gorton, A. Fekete, Design-level performance prediction of component-based applications, IEEE Trans. on Software Engineering 31 (11) (2005) 928–941.
[8] P. Merson, S.A. Hissam, Predictability by construction, in: SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05), San Diego, CA, October, ACM, 2005.
[9] S.A. Hissam, G.A. Moreno, K.C. Wallnau, Using Containers to Enforce Smart Constraints for Performance in Industrial Systems, CMU/SEI-2005-TN-040, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, 2005.
[10] G.A. Moreno, Creating custom containers with generative techniques, in: Proc. 5th International Conference on Generative Programming and Component Engineering (GPCE06), Portland, OR, Oct., 2006.
[11] L. Bass, J. Ivers, M. Klein, P. Merson, Reasoning Frameworks. CMU/SEI-2005-TR-007. Software Engineering Institute - Carnegie Mellon University. Pittsburgh, PA, 2005.
[12] J. Ivers, G.A. Moreno, Model-driven development with predictable quality, in: SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA07), Montreal, Quebec, Canada, October, ACM, 2007.
[13] K.C. Wallnau, J. Ivers, Snapshot of Ccl: A Language for Predictable Assembly, CMU/SEI-2003-TN-025. Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, 2003.
[14] S.A. Hissam, J. Ivers, D. Plakosh, K.C. Wallnau, Pin Component Technology (V1.0) and Its C Interface, CMU/SEI-2005-TN-001, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, 2005.
[15] M. Gonzalez Harbour, J.J. Gutierrez Garcia, J.C. Palencia Gutierrez, J.M. Drake Moyano, Mast: Modeling and analysis suite for real-time applications, in: Proceedings 13th Euromicro Conference on Real-Time Systems (ECRTS), Washington, DC, June, IEEE Computer Society, 2001.
[16] C.U. Smith, 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), Enschede, NL, Sept. 2004, IEEE Computer Society, 2004, pp. 38–47.
[17] C.U. Smith, C.M. Lladó, V. Cortellessa, A. Di Marco, L.G. Williams, From Uml models to software performance results: An Spe process based on Xml interchange formats, in: Workshop on Software and Performance (WOSP05), Palma de Mallorca, July, ACM, 2005, pp. 87–98.
[18] L.G. Williams, C.U. Smith, Information requirements for software performance engineering, in: Proceedings 1995 International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Heidelberg, Germany, Sept., Springer, 1995.
[19] C.U. Smith, L.G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, Boston, 2002.
[20] L&S, Computer Technology, Inc., Performance Engineering Services Division, # 110, PO Box 9802, (505) 988-3811, www.spe-ed.com, Austin, TX 78766.
[21] C.U. Smith, C.M. Lladó, L.G. Williams, R. Puigjaner, Interchange formats for performance models: Experimentation and output, in: Proc. Quantative Evaluation of Systems (QEST), Edinburgh, Scotland, Sept., IEEE, 2007.
[22] W3C, World Wide Web Consortium, www.w3c.org, 2001.
[23] S. Kent, Model driven engineering, in: M.J. Butler, L. Petre, K. Sere (Eds.), Proc Third Int. Conf. On Integrated Formal Methods, in: Lecture Notes in Computer Science, Springer-Verlag, London, 2002, pp. 286–298.
[24] D.C. Schmidt, Guest editor's introduction: Model-driven engineering, IEEE Computer 39 (2) (2006) 25–31.
[25] OMG, Meta Object Facility (Mof) 2.0 Core Specification, http://www.omg.org/mof/, 2004.
[26] OMG, Modeling and Analysis of Real-Time and Embedded Systems (Marte), http://www.omgmarte.org/, 2007.
[27] R. Kazman, G. Abowd, L. Bass, P. Clements, Scenario-based analysis of software architecture, IEEE Software 13 (6) (1996) 47–55.
[28] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, in: International Conference on Engineering of Complex Computer Systems (ICECCS98), Aug., 1998.
[29] L.G. Williams, C.U. Smith, Pasa^{sm}: A method for the performance assessment of software architectures, in: Proc. 3rd Int. Workshop on Software and Performance, Rome, IT, July, ACM Press, 2002.
[30] S. Balsamo, P. Inverardi, C. Mangano, An approach to performance evaluation of software architectures, in: Workshop on Software and Performance, Santa Fe, NM, October, ACM, 1998, pp. 178–190.
[31] S. Balsamo, M. Marzolla, Performance evaluation of Uml software architectures with multiclass queueing network models, in: WOSP 2005, Palma de Mallorca, July, ACM, 2005, pp. 37–42.
[32] G. Gu, D. Petriu, From Uml to Lqn by Xml algebra-based model transformations, in: WOSP 2005, Palma de Mallorca, July, ACM, 2005, pp. 99–109.
[33] J.P. López-Grao, J. Merseguer, J. Campos, From Uml activity diagrams to stochastic Petri nets: Application to software performance engineering, in: Proc. Workshop on Software and Performance, Redwood Shores, CA, Jan., ACM, 2004, pp. 25–36.
[34] D. Petriu, C.M. Woodside, Analyzing software performance requirements specification for performance, in: Proc. Workshop on Software and Performance 2002, Rome, July, ACM, 2002, pp. 1–9.
[35] N. Savino, E. Arraiz, A. Di Serio, J.L. Anciano, R. Puigjaner, 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, San Diego, CA, 2002.
[36] C.U. Smith, C.M. Lladó, Performance Model Interchange Format (Pmif 2.0): Xml Definition and Implementation Update Technical Report. L&S Computer Technology, Inc. Santa Fe, NM, 2007.
[37] C.U. Smith, V. Cortellessa, A. Di Marco, C.M. Lladó, L.G. Williams, From Uml models to software performance results: An Spe process based on Xml interchange formats, in: Proc. 5th Int. Workshop on Software and Performance, Palma, Illes Balears, Spain, July, ACM Press, 2005, pp. 87–98.
[38] C.M. Woodside, D. Petriu, D. Petriu, H. Shen, T. Israr, J. Merseguer, Performance by unified model analysis (Puma), in: WOSP 2005, Palma de Mallorca, July, ACM, 2005, pp. 1–12.
[39] A. D'Ambrogio, A model transformation framework for the automated building of performance models from Uml models, in: Proc. 2005 Workshop on Software and Performance, Palma de Mallorca, July, ACM Press, 2005, pp. 75–86.
[40] V. Grassi, R. Mirandola, A. Sabetta, A model transformation approach for the early performance and reliability analysis of component-based systems, in: Lecture Notes in Computer Science, vol. 4063, 2006, pp. 270–284.
[41] H. Koziolek, R. Reussner, A model transformation from the Palladio Component Model to layered queueing networks, in: Lecture Notes in Computer Science, vol. 5119, 2008, pp. 58–78.
[42] A. Bertolino, V. Cortellessa, A. Di Marco, R. Mirandola, From Uml to Execution Graphs and Queueing Networks: Design and Implementation of the Xml-Based Tool Xprimat. Universita del L'Aquila. L'Aquila, Italy, 2004.
[43] V. Cortellessa, M. Gentile, 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, Enschede, NL, IEEE Computer Society, 2004, pp. 342–343.
[44] H. Gomaa, D.A. Menasce, Performance engineering of component-based distributed software systems, in: Dumke, et al. (Eds.), Lncs 2047: Performance Engineering State of the Art and Current Trends, Springer-Verlag, Berlin, 2001, pp. 40–55.
[45] G. Gu, D. Petriu, Xslt transformation from Uml models to lqn performance models, in: Proc. Workshop on Software and Performance, Rome, July, 2002, ACM, 2002, pp. 227–234.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008

[46] V. Cortellessa, S. Di Gregorio, A. Di Marco, Using atl for transformations in software performance engineering: A step ahead of java-based transformations? in: Proc. Workshop on Software and Performance, Princeton, NJ, June, ACM Press, 2008, pp. 127–131.
[47] S. Becker, Coupled model transformations, in: Proc. Workshop on Software and Performance, Princeton, NJ, June, ACM Press, 2008, pp. 103–114.
[48] A. Gherbi, F. Khendek, From Uml/Spt Models to Schedulability Analysis: Approach and a Prototype Implementation, Automated Software Engineering http://www.springerlink.com/content/jt45l5550m3u6p72/, 2009.
[49] X. Wu, C.M. Woodside, Performance modeling from software components, in: Proc. Workshop on Software and Performance, Redwood Shores, CA, January, ACM, 2004, pp. 290–301.
[50] S. Becker, H. Koziolek, R. Reussner, Model-based performance prediction with the Palladio Component Model, in: Workshop on Software and Performance (WOSP07), Buenos Aires, Argentina, Feb., ACM, 2007.
[51] V. Grassi, R. Mirandola, A. Sabetta, A model-driven approach to performability analysis of dynamically reconfigurable component-based systems, in: Workshop on Software and Performance (WOSP07), Buenos Aires, Argentina, Feb., ACM, 2007.
[52] V. Cortellessa, How far are we from the definition of a common software performance ontology? in: WOSP 2005, Palma de Mallorca, July, ACM, 2005, pp. 195–204.
[53] C.U. Smith, Performance Engineering of Software Systems, Addison-Wesley, Reading, MA, 1990.
[54] EDIF. Edif Users' Group. Design Automation Department, Texas Instruments, PO Box 225474, MS-3668, Dallas TX 75265.
[55] EIA. 1994. Cdif - Case Data Interchange Format Overview. EIA/IS-106. Engineering Department, Electronics Industries Association, Arlington, VA.
[56] C.U. Smith, L.G. Williams, Performance engineering of object-oriented systems with speed, in: R. Marie, et al. (Eds.), in: Lecture Notes in Computer Science 1245: Computer Performance Evaluation, Springer, Berlin, Germany, 1997, pp. 135–154.
[57] C.U. Smith, L.G. Williams, Performance engineering evaluation of corba-based distributed systems with Spe-Ed, in: R. Puigjaner (Ed.), in: Lecture Notes in Computer Science, Springer, Berlin, Germany, 1998.
[58] P. Hughes, Sp Principles, O59/ICL226/0, STC Technology, 1988.
[59] H. Beilner, J. Mäter, N. Weissenburg, Towards a performance modeling environment: News on hit, in: Proceedings 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Plenum Publishing, 1988.
[60] C.U. Smith, C.M. Lladó, R. Puigjaner, Automatic generation of performance analysis results: Requirements and definition, Lecture Notes in Computer Science: Proceedings of the European Performance Evaluation Workshop, 2009.
[61] M. Gonzalez Harbour, M. Klein, J. Lehoczky, Timing analysis for fixed-priority scheduling of hard real-time systems, IEEE Trans. on Software Engineering 20 (1) (1994) 13–28.

Gabriel A. Moreno received the BS degree in Computing Systems from University of Mendoza, Argentina, and the Masters of Software Engineering degree from Carnegie Mellon University. He is a Senior Member of the Technical Staff at the Carnegie Mellon Software Engineering Institute. Previously, he was at ITC Soluciones, Argentina, where he designed and developed multiplatform distributed systems and communication protocols for electronic transactions. He has received several awards, including a Fulbright Fellowship. His research interests include predictable assembly, performance modeling and analysis, component technology, and model-driven development.

Connie U. Smith, is a principal consultant of the Performance Engineering Services Division of L&S Computer Technology, Inc. She received a BA in mathematics from the University of Colorado and MA and Ph.D. degrees in computer science from the University of Texas at Austin. She is the author of *Performance Engineering of Software Systems*, and co-authored the *Performance Solutions* book, and numerous scientific papers. She is the principal developer of the performance engineering tool, *SPE· ED*™. Her research interests include computer performance modeling and evaluation, tool interoperability, and software engineering. Dr. Smith received the Computer Measurement Group's AA Michelson Award for technical excellence and professional contributions for her SPE work. She frequently serves on conference and program committees, and currently chairs the steering committee of the Workshop on Software and Performance.

Please cite this article in press as: G.A. Moreno, C.U. Smith, Performance analysis of real-time component architectures: An enhanced model interchange approach, Performance Evaluation (2009), doi:10.1016/j.peva.2009.07.008