EVALUATING QUALITY OF SERVICE FOR ENTERPRISE DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

James H. Hill
hillj@dre.vanderbilt.edu
Institute for Software Integrated Systems
2015 Terrace Place
Nashville, TN, 37203, USA
+1 615-343-8197

Douglas C. Schmidt
schmidt@dre.vanderbilt.edu
Institute for Software Integrated Systems
2015 Terrace Place
Nashville, TN, 37203, USA
+1 615-343-8197

John M. Slaby
john_m_slaby@raytheon.com
Raytheon Integrated Defense Systems
1847 West Main Road
Portsmouth, RI 02871
+1 401-842-2107

EVALUATING QUALITY OF SERVICE FOR ENTERPRISE DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

This chapter introduces the next generation of system execution modeling tools designed around model-driven engineering (MDE) coupled with domain-specific modeling languages (DSMLs). The authors discuss key design issues involved with implementing a next generation SEM tool and show how they can be applied to developing service-oriented architecture (SOA)-based applications. Finally, the authors use a real-life case study to illustrate how next generation system execution modeling tools can help understand quality-of-service (QoS) issues earlier in the development lifecycle (i.e., during design-time) instead of waiting until complete system integration.

Keywords: CASE tools, Distributed Systems, Input/Output Models, Modeling Languages, Process Model, Structural Modeling, Systems Evaluation

INTRODUCTION

Integration Challenges of SOA-based Enterprise DRE Systems

Enterprise *distributed real-time and embedded* (DRE) systems, such as supervisory control and data acquisition (SCADA) systems, air traffic control systems, and shipboard computing environments, are growing in complexity and importance as computing devices are networked together to help automate tasks previously done by human operators. These types of systems are required to provide quality of service (QoS) support to process the right data in the right place at the right time over a networked grid of computers. QoS properties required by enterprise DRE systems include the low latency and jitter expected in conventional real-time and embedded systems, and the high throughput, scalability, and reliability expected in conventional enterprise distributed systems. Achieving this combination of QoS capabilities is hard because these systems work in constrained environments with a limited amount of resources that can vary depending on the location of the system. Moreover, this level of QoS requires in depth knowledge of low-level programming techniques, e.g., properly interfacing with sockets to write efficient networking protocols, which the application developers of enterprise DRE system may not possess.

To address these challenges, enterprise DRE systems are increasingly being developed using applications composed of components running on feature–rich *service-oriented architecture* (SOA) middleware frameworks. These components are designed to provide reusable services to a range of application domains that are composed into domain-specific assemblies for application (re)use. SOA middleware is intended to alleviate problems of inflexibility and reinvention of core capabilities associated with prior monolithic, functionally-designed, and "stove-piped" legacy applications developed using just the capabilities required for a specific set of requirements and operating conditions. SOA-based systems, conversely, are designed to have a more general range of capabilities that enable their reuse in other contexts. Moreover, these systems are developed in layers, e.g., layer(s) of infrastructure middleware services (such as naming and discovery, event and notification, security and fault tolerance) and layer(s) of application components that use these services in different compositions.

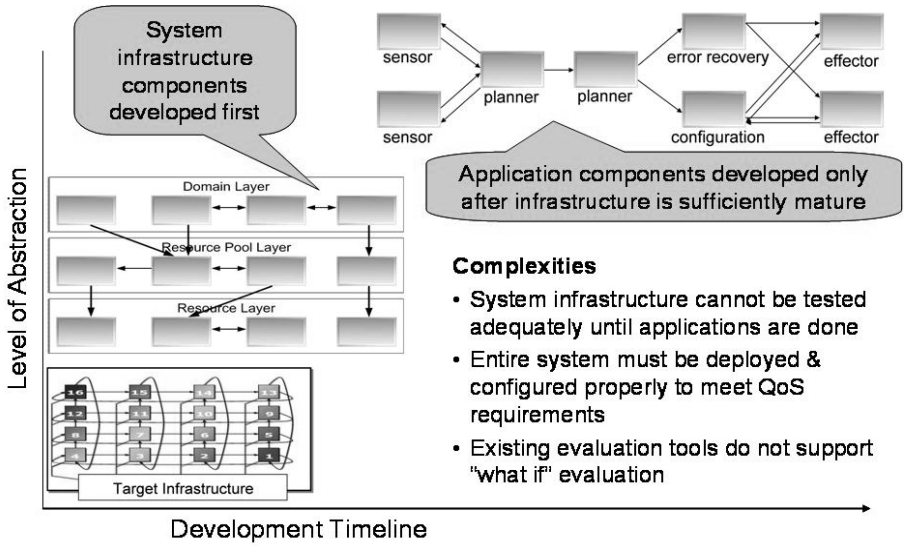

Figure 1. Characteristics and Complexities of Serialized Phasing in Enterprise DRE Systems

Combining stringent QoS requirements in DRE systems with the transition to SOA component frameworks has created a particularly vexing problem for researchers and developers of large and layered enterprise DRE systems: the inadequacies of system architectures may not be ascertained until years into development. At the heart of this problem is the *serialized phasing* of layered system development, shown in Figure 1, which postpones the discovery of design flaws that affect system QoS until late in the lifecycle, i.e., at integration time. A hallmark of serialized phasing is that application components are not created until *after* their underlying system infrastructure components, such as naming and discovery, event and notification, security and fault tolerance, and resource management.

As shown in Figure 1, SOA-based enterprise DRE systems built using serialized phasing often do not adequately test the implementations, configurations, and deployments of infrastructure components under realistic workloads until the application components are done. Moreover, both application and infrastructure components are hosted on the same target platform. Each component must, therefore, be properly deployed and configured to achieve the desire QoS. As a result, it hard to know how well the system will satisfy key QoS properties due to disconnects in the phasing of infrastructure and application component development. Moreover, handcrafted software designs used in many enterprise DRE systems to address these concerns make it hard to conduct "what if" experiments on alternative system architectures and implementations to determine which valid configurations can obtain performance goals for a particular workload. Making any significant changes to these types of handcrafted systems late in their lifecycle can be costly due to the impact on the design, implementation, deployment, and (re)validation of many application and infrastructure software/hardware components.

Solution Approach: **System Execution Modeling Tools**

To address the problems in SOA-based enterprise DRE systems, there is a need for a methodology and an associated suite of *system execution modeling* (SEM) tools that use *model-*

driven engineering (MDE) (Schmidt, 2006) technologies, such as GME (Karsai, 2003) or GEMS (White, 2007), to simplify the:

1. **Emulation of application component behavior** in terms of computational workloads, resource utilizations and requirements, and network communication. This step can be accomplished quickly and precisely using domain-specific modeling languages (DSMLs) (Ledeczi, 1999) that capture the behavior and workload of system components (at a higher-level of abstraction than third-generation languages like C++ or Java). DSML interpreters then parse the constructed behavior and workload models to generate code that executes emulated components.

2. **Configuration, deployment, and execution** of the emulated application components atop actual infrastructure components to determine their impact on QoS empirically in actual runtime environments. These steps can also be accomplished using DSMLs that specify realistic deployments and configurations and then generate the associated metadata describing these deployments and configurations. These metadata descriptions are processed by the same deployment and configuration tools as the final system, with SEM tools providing mechanisms to record, consolidate, and collect QoS metrics (such as execution times and resource usage) from the SOA runtime environment.

3. **Process of feeding back the results** to enhance system architectures and components to improve QoS. This step can be accomplished by archiving the collected QoS metrics and providing tools that view the overall results of a deployment. SEM tools also provide histories of the collected metrics to enable engineers and architects to understand end-to-end system performance and make well-informed decisions on how to improve QoS.

As actual application components mature over time, they can replace the emulated components, thereby providing an ever more realistic evaluation environment. Figure 2 shows the relationships between the steps described above.

SEM tools enable system engineers, software architects/developers, and quality assurance (QA) engineers to address the inherent complexities that arise from properties of production systems, including communication delay, temporal phasing, parallel execution, and synchronization. There are typically only a few execution designs that actually can satisfy the functional and performance requirements established in software and system architecture. SEM tools enable architects and engineers to discover, measure, and rectify incipient integration and performance problems early in a system's lifecycle (e.g., in the analysis and/or design phases). These tools help shift the focus of the software integration resources to productive activities that evaluate and validate system performance and end-user value, rather than serving as the *de facto* system design debugging activity, as is often the case today.

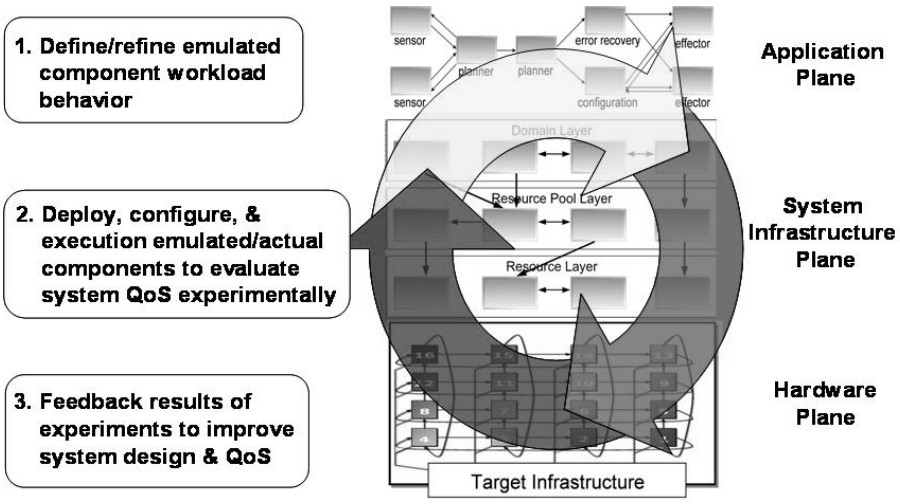

Figure 2. Evaluating the QoS of Enterprise DRE System via System Execution Modeling Tools

This chapter illustrates by example the following concepts for using MDE-based SEM tools to evaluate SOA-based enterprise DRE systems:

- The purpose of next-generation SEM tools and the limitations they address with traditional SEM tools.
- The different elements needed to construct a next-generation SEM tool, including designing the behavioral and workload DSMLs to capture emulated behavior, selecting the method(s) for configuring, deploying, and executing emulated system behavior, and understanding different analytical techniques to feedback results to users.
- The application of next-generation SEM tools to SOA-based enterprise DRE systems to evaluate system QoS to show by example how SEM tools can be used to evaluate enterprise DRE systems during the early stages of development, i.e., before complete system integration.

BACKGROUND

Before we begin our discussion on MDE-based SEM tools, we first describe existing techniques and tools used to evaluate the QoS of enterprise DRE systems. This section summarizes conventional techniques and tools for evaluating enterprise DRE system performance in different phases of development.

Distributed System Emulation Testbeds

During the past several years a number of testbeds have been developed for emulating and evaluating the behavior of distributed systems in networked environments. One such testbed is Emulab (Ricci, 2003), which originated at the University of Utah to provide freely available resources and tools to configure the topology of experiments, e.g., modeling the underlying communication links. The virtual topology is then mapped to ~250 physical nodes that can be accessed and managed via the Internet. Faux and/or real applications can be executed in this environment to evaluate the performance of both the topology and applications. As a result of Emulab's success, other institutions and organizations, such as Cornell University,

Georgia Institute of Technology, and Vanderbilt University, are hosting their own Emulab for private and/or public use.

Another common testbed is PlanetLab (2006), which is managed by Princeton University, the University of California Berkeley, and the University of Washington. PlanetLab provides a similar user experience as Emulab, though it focuses on large-scale (Northrop, 2006) distributed systems. PlanetLab currently consists of 726 machines, hosted by 354 sites, spanning over 25 countries. Most machines are hosted by research institutions, but regardless of where the machine is hosted, it is accessible via the Internet. Researcher can request *slices* of PlanetLab to experiment with a variety of planetary-scale services, such as content distribution networks, QoS overlays, scalable event propagation, anomaly detection mechanisms, and network measurement tools, to run experiments. The goal for PlanetLab is to grow to over 1,000 widely distributed nodes that peer with a majority of the Internet's regional and long-haul backbones.

ModelNet (Vahdat, 2002) is another testbed for emulating and evaluating large-scaled distributed systems. With ModelNet, developers can emulate multiple clients and hosts using a single physical host. ModelNet provides similar functionality as Emulab, though it focuses on resource constrained environments, i.e., environments that do not have access to enough resources to scale to the deployment environment. For example, 100 Gnutella clients each with a 1 Mbps bottleneck bandwidth can be emulated on one dual processor-1 GHz machine. In addition to providing a scalable emulation environment, ModelNet facilitates the emulation of faux and real applications.

Existing distributed system emulation testbeds are useful in the early stages of development when testing functionality under various conditions/configurations, especially when the target platform is not known *a priori*. With enterprise DRE systems, however, the target platform(s) are usually known (and available) at the start of development. What is needed, therefore, are next-generation SEM tools that will help developers and testers leverage the same benefits provided by existing (public) testbeds to run as many experiments as possible on the target platform, which is usually a private-based testbed built to the specifications of the target project.

System Execution Modeling (SEM) Tools

Performance evaluation of systems has always been a research topic that has received much attention. For example, Smith (1990, 2001) has shown how variations of queuing theory (Denning, 1978) can be applied to evaluate the performance of enterprise systems. The result of her work lead the creation of a SEM tool designed specifically for software performance evaluation (SPE) called *SPE·ED* (www.perfeng.com/sped.htm). SPE·ED allows developers to model the "business-logic" of their system and parameterize the model with performance metrics, e.g., arrival rate and throughput of events, service rates of devices, and resource availability. Testers can then run simulations of their modeled system and analyze its "expected" performance. Analysis results can include determining the maximum throughput for each device, locating the bottleneck device in the system, or evaluating performance under expected, or hypothetical, system upgrades.

UPPAAL (Bengtsson, 1995) is an integrated tool environment for modeling, simulating, and verifying real-time systems developed jointly by Basic Research in Computer Science at Aalborg University in Denmark and the Department of Information Technology at Uppsala University in Sweden. It is based on the formal language of timed automata (Subramonian, 2006), but does not require in-depth knowledge for basic usage. Similar to most SEM tools, UPPAAL provides a graphical interface to simplify the creation of timed automata models for the system under development. More importantly, the graphical interface also allows developers to visualize the simulations of the system under development. Lastly, the constructed models can be run through a model checker to check invariant and reachability properties by exploring the state-space of a system.

CPN Tools (2006) is another SEM tool that allows developers to capture the *behavior* of a component, or system under development, using color Petri nets (Kristensen, 1998). Similar to UPPAAL, CPN Tools provides a graphical user interface to simplify the creation of color Petri net models. CPN Tools, however, requires some level of expertise and understanding of color Petri nets. Once models are constructed using CPN Tools, they can be simulated to verify different properties of DRE system, such as correctness and state reachability, which is similar to UPPAAL. Performance metrics, e.g., service time, arrival rate, or resource utilization, can also be associated with the states and transitions in the models to run SPE simulations, which is similar to SPE·ED.

Other modeling languages, such as KLAPER (Grassi, 2005) and RT-UML (Bertolino, 2004), can be used to model system execution. KLAPER is a modeling language that facilitates workload specifications, such as resource utilization, which is then emulated in its own proprietary tool. RT-UML models and evaluates the performance of component-based systems by defining services and QoS policies for components; however, modeling system behavior is future work. RT-UML is also designed to be supported by external *simulation* tools.

Many of the tools discussed above can also be applied in the area of soft and hard real-time systems. In these types of systems, however, more focus is placed on completing tasks in a given time constraint, as opposed to verifying the state of the system, e.g., the current values of attributes or utilizations of resources. In hard real-time systems, developers are concerned with achieving worst case execution time within a specified time constraint, whereas in soft real-time systems developers are concerned with achieving average execution time within a specified time constraint at a given probability (Florescu, 2006).

Tavares et al. (Tavares, 2005) demonstrates how time Petri nets (Merlin, 1974) can be applied to verify the scheduling of hard real-time tasks when considering multiple system constraints, such as execution time and power consumption. Likewise, Bucci et al. (Bucci, 2003) illustrates how preemptive time Petri nets, which extend time Petri nets, can be used to verify the scheduling of hard real-time tasks set with flexible computations, such as periodic, sporadic, and non-deterministic execution times. In the domain of soft real-time systems, Florescu et al. (Florescu, 06) introduces an approached called *probabilistic modeling and evaluat*ion for soft real-time systems. This approach uses a language called Parallel Object-Oriented Specification Language (POOSL) (www.es.ele.tue.nl/poosl) and involves modeling the distribution of a task's measured execution times over a given period of time. The con-

structed model is then analyzed to understand the probability of the task achieving a specific execution time in the future given its measured distribution curve of execution times.

It is clear that existing SEM tools are useful for understanding the state space of a system by providing high-level abstractions (i.e., conceptual models of system) that shield developers from low-level implementation details. Existing SEM tools, however, do not provide support for enterprise DRE systems developed using SOA technologies, which are developed primarily using DSML tools and not generalized tools. What is needed, therefore, are next-generation SEM tools that can provided the same functionality as existing SEM tools, but are tailored for SOA-based enterprise DRE systems, i.e., provide the metadata generation capability of MDE tools and the evaluation capabilities of existing SEM tools.

Evaluation Techniques for Component Architectures

There are several techniques for evaluating the performance of component architectures, including *event tracing* and *system profiling*. Event tracing techniques are typically based on observing the performance of a single event (e.g., execution path and time) as it travels through the system (i.e., transmitted end-to-end from component to component). System profiling techniques often use external tools to monitor the performance of software and hardware while the system is executing and transcribe collected metrics to files for analysis once the system is offline.

Mania (2002) discusses a technique called *trace-based analysis* for Enterprise Java Bean (EJB) components. In trace-based analysis, different execution traces, i.e., function calls, are monitored and outputted to a trace file contained on the host. After the emulation, trace files are parsed and combined with the deployment descriptors to determine the different paths of execution in the system.

Hauswirth (2005) discusses *vertical profiling* evaluation techniques in the context of EJB. In vertical profiling, performance metrics based on the types of operations and actions, e.g., cache misses and CPU cycles, are collected in trace files. Trace files are then fused through a process called *trace-alignment* using a common metric that occurs in the source traces. After the traces are aligned, *correlation analysis* is applied to the traces to help determine what other metrics collected in the trace may influence its behavior.

Existing techniques for evaluating the performance of component architectures are relatively complex and low-level, i.e., at the middleware infrastructure level. What is needed, therefore, are next-generation SEM tools that provide the same analysis techniques, but shield the developer from the complexity of existing tools. Moreover, these tools should allow developers to leverage existing MDE tools for component architectures, but provide feedback to pinpoint how the low-level analysis correlates to the high-level component architecture's implementation.

MOTIVATING EXAMPLE AND CASE STUDY

To motivate the structure and functionality of next-generation SEM tools, this section presents a case study from the domain of enterprise DRE systems. This case study focuses on a SOA-based *multi-layer resource management* (MLRM) infrastructure (Lardieri, 2007) for

naval shipboard computing systems and the challenges encountered while developing and evaluating it. The MLRM service architecture shown in Figure 3 forms the basis for future naval programs, which run on a coordinated grid of computers that manage many aspects of a ship's power, navigation, command and control, and tactical operations. Our motivating example is from the domain of naval shipboard computing, and more specifically a closed system. We believe, however, that the structure and functionality of next-generation SEM tools can also be applied to open systems, such as peer-to-peer applications and service providers systems (e.g., online stock applications) (Hill, 2007).

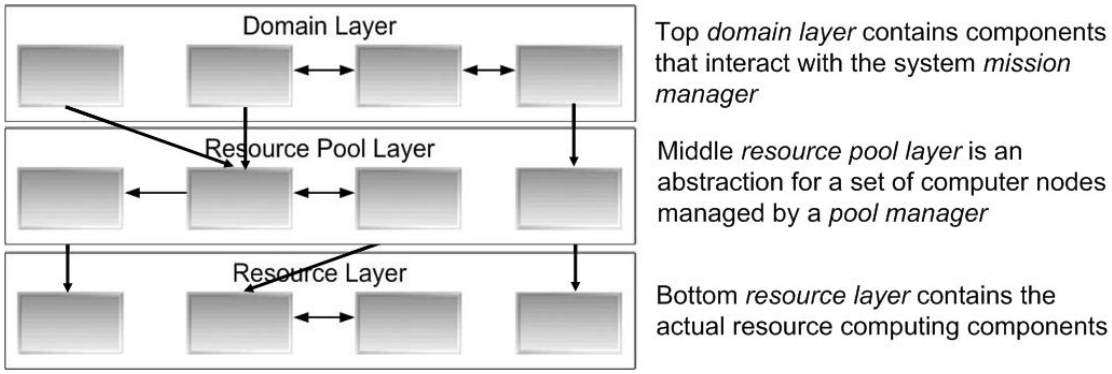

Figure 3. SOA-based Multi-Layer Resource Manager (MLRM) Infrastructure for Shipboard Computing

The MLRM, shown in Figure 3, consists of the three layers. The command and policy inputs flow in a top-down manner and correspondingly the resource status information moves in a bottom-up fashion. At the top is the *domain layer*, which contains infrastructure components that interact with the mission manager of shipboard environment by receiving command and policy inputs and passing them to the *resource pool layer*. The resource pool layer is an abstraction for a set of computer nodes managed by a *pool manager*. The pool manager is an infrastructure component that interacts with the *resource allocator* in the resource pool layer to run algorithms that deploy application components to various nodes within a resource pool. The actual computing resources reside in the third layer called the *resource layer*, which has infrastructure components called *node provisioners* that receive commands to spawn applications in every node from a pool manager.

The SOA-based MLRM services described above are designed to support the highly heterogeneous environment in which long-lived enterprise DRE systems operate. For example, the Naval program that provides the operational context for the MLRM services is designed to support different versions of (1) component middleware, such as CIAO and OpenCCM, (2) general-purpose operating systems, such as Linux and Solaris, (3) real-time operating systems, such as VxWorks and LynxOS, (4) hardware chipsets, such as x86, PowerPC, and SPARC processors, (5) a wide range of high-speed wired interconnects, such as Gigabit Ethernet and Infiniband, and (6) different transport protocols, such as TCP/IP and SCTP.

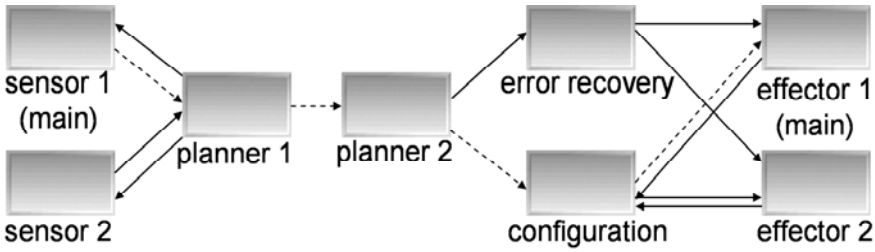

Figure 4. Model of SLICE Showing the MLRM Components and Their Interconnections

Figure 4 shows one of the challenge problems of the MLRM case study called the *SLICE scenario*, which consists of 2 sensors, 2 planners, 1 configuration, 1 error recovery, and 2 effector components. This scenario requires the transmission of information detected by the sensors to each planner in sequence, then to the configuration component, and lastly to both effectors to perform actions that control devices in the physical world. Components in the SLICE scenario are deployed across 3 computing nodes because the workload generated by the components collectively is more than a single node can handle. The main sensor and effector (represented as sensor-1 and effector-1 in Figure 4 and in following discussions) are deployed on separate nodes to reflect the placement of physical equipment in the production shipboard system. A model of the end-to-end layout of SLICE components is shown in Figure 4, with the *critical path* (i.e., sequence of components that must meet a predetermined end-to-end deadline) specified by the dashed arrows.

Based on the MLRM development schedule, the integration of components that implement the SLICE scenario atop the multi-layer resource management infrastructure was not projected to occur until 12 months into the program to provide sufficient time to finish developing, testing, and optimizing it. Since these components were currently under development, however, we understood each component's behavior and resource usage expectations in SLICE. What we did *not* know was how the overall performance of the SLICE scenario would be affected when deployed with the MLRM infrastructure.

In a conventional project developed with serialized phasing, we would have waited until final system integration to benchmark the entire system. If integration testing revealed problems with the MLRM infrastructure, the process of reconfiguring and redeploying application and infrastructure components to meet QoS requirements would have required significant effort. Moreover, developers and testers would have to use existing evaluation techniques to locate problematic areas and manually pinpoint their correlation in MLRM's implementation. Developers would also have to continuously revise *completed* infrastructure code without knowing how the changes will affect application-level performance. To prevent reimplementation late in the development cycle (e.g., at integration time) we could use the tools like UPPAAL and CPN at early stages of development to *predict* the expect performance of the system. Although an analytic understanding of performance based on simulation is usually better than no understanding at all, these tools have the following limitations that make them inadequate for accurately evaluating the QoS of enterprise DRE systems, such as the SLICE scenario:

- They do not execute in the actual target environment, which precludes system testers from producing "realistic" performance results based the real hardware and software con-

figuration. Moreover, existing SEM tools do not take into account non-deterministic behavior that can be introduced by component architectures, such as reliable communication and security. Subramonian (2006) has done work to extend existing SEMS tools to handle enterprise DRE systems, but these techniques require a high degree of user expertise.

- They are not designed to integrate seamlessly with contemporary SOA platforms, e.g., they require developers to learn "low-level" techniques (such as observing cache misses, disk access time, or disk utilization) when they are developing at a "high-level" of abstraction (such as the application logic). Moreover, users must understand how to correlate the "low-level" performance metrics to "high-level" implementation.

The remainder of this chapter focuses on building and applying next-generation MDE-based SEM tools to help address the challenges of evaluating SLICE scenario performance at early stages of development. The goals of this case study are to (1) simplify the process of determining which deployment and configuration strategies will meet critical path QoS deadlines, (2) create a catalog of selectable deployment strategies that meet end-to-end performance requirements, and (3) spend less time integrating and testing the actual SLICE components after they are completed, i.e., to reduce time spent in system integration, which still ensuring that QoS requirements are met.

APPLYING NEXT-GENERATION SEM TOOLS TO SOA-BASED DRE SYSTEMS

In the introduction, we described the need for next-generation MDE-based SEM tools to simplify the development of SOA-based enterprise DRE systems. To address the complexities of existing tools and techniques discussed in System Execution Modeling (SEM) Tools Section requires developing the necessary MDE infrastructure for next-generation SEM tools. Figure 5 illustrates the elements and workflow of one such architecture called CUTS (Slaby, 2006).

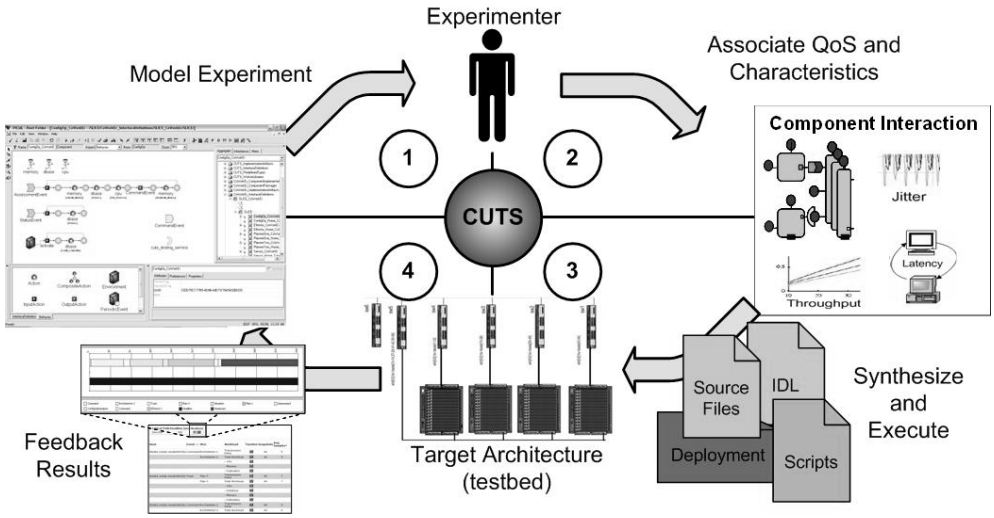

Figure 5. Architecture and Workflow for Next-generation SEM Tools

At the heart of the architecture in Figure 5 is the *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS)*, which is based on the CoSMIC (Gokhale, 2006) MDE tool chain. This figure shows the following steps:

1. In this step users (e.g., software architects, developers, and systems engineers) specify the structure of an enterprise DRE system (e.g., the component and their interconnections using CUTS DSMLs).
2. In this step users can associate the necessary QoS characteristics with individual components (e.g., CPU utilization) or the system as a whole (e.g., deadline of a critical path through the system).
3. In this step the information captured by the tools can be synthesized into executable code and configuration metadata, which the middleware then uses to deploy the emulated/actual application/system components onto the target platform.
4. In this step system developers and engineers analyze the collected metrics and explore design alternatives from multiple computational and valuation perspectives to quantify the costs of certain design choiceson end-to-end system performance.

This process can be applied iteratively throughout the phases of development process.

In the context of SOA-based enterprise DRE systems, the CUTS SEM tools helps developers conduct "what if" experiments to discover, measure, and rectify performance problems *early* in the lifecycle (e.g., in the architecture and design phases), as opposed to the integration phase, when mistakes are much harder and more costly to fix. The remainder of the section discusses the elements (i.e., the modeling languages, emulation methods, and analysis methods) needed to create a SEM tool chain using MDE technologies. For each element we describe the problems faced and solutions applied in the context of CUTS and the SLICE scenario.

Emulating Application Behavior using Modeling Languages

Context. When using an MDE tool to develop a SOA-based enterprise DRE system it is necessary to capture the structure of the system, i.e., the components' interfaces and attributes, the interconnections between components, and the behavior. Many MDE tools, such as Cadena (Hatcliff, 2003), PICML (Balasubramanian, 2005), and J2EEML (White, 2005), capture structural aspects of SOA-based systems rather than the behavioral aspects. The emphasis on structure aspects stems from the fact that enterprise DRE systems built using SOA technologies like J2EE, CCM, or Microsoft .NET depend heavily on XML-based descriptor files. A straightforward use of an MDE tool, therefore, is to auto-generate XML files that are tedious and error-prone to handcraft manually.

Problem → Capturing system behavior using modeling languages. Capturing the behavior of a component to perform early design-time analysis before the integration phase requires developers to rely on external tools and languages since existing MDE technologies often do not provide the same capabilities as existing behavior analysis tools and languages. For example, external tools, such as Petri nets, UPPAAL, and CPN Tools allow developers to model the behavior of the system under development and run simulations to analyze its expected performance. Likewise, languages such as SIMULA and Z allow developers to write

simulations of systems for verification purposes. External dependencies on these tools and languages, however, make it hard to map the results from design-analysis tools to the respective areas in structure models provided by the MDE tool.

Existing behavior analysis tools and languages are also geared largely to software developers with deep knowledge of formal methods and advanced mathematical formalisms. In particular, it requires these developers to learn and manage multiple languages and tools, which can be hard if the knowledge base in not available. These tools and languages are generally not usable by key participants throughout the software lifecycle of enterprise DRE systems, including subject matter experts, systems engineers, and quality engineers

Solution → Integrate behavior models into existing MDE tools. Ideally, the DSML in a SEM tool for behavioral modeling should capture both actions and workload. The DSML interpreter should also generate the necessary files (e.g., source code and configuration files) that use proprietary methods or third-party tools to perform early design-time analysis. The advantage of this approach versus using separate tools for capturing structure and behavior is that it reduces the number of tools the developers and experimenters have to explicitly manage. Moreover, an integrated approach can help correlate performance results with the appropriate parts of the target model. Lastly, this approach removes the complexity of manually handcrafting the source and configuration files to use existing analysis tools and languages since they can be auto-generated directly from the integrated model.

Applying the solution to CUTS and SLICE. When developing CUTS we created two DSMLs—the *Component Behavior Modeling Language* (CBML) and the *Workload Modeling Language* (WML) (Hill, 2007)—and integrated them into the *Platform Independent Component Modeling Language* (PICML) in CoSMIC. CBML is a modeling language based on I/O automata (Lynch, 1989) (see Sidebar 1) that allows developers to capture a component's internal actions. Although CBML is based on a mathematical formalism, its users need not have expertise in the low-level details of I/O automata programming since it provides an MDE-based interface, as opposed to a traditional text-based interface. As a result CBML simplifies the use of I/O automata, such as auto-generating the model elements for modelers or generating text-based configuration files for I/O automata tools, such as the TIOA Language and Toolset (Garland, 2005).

WML is an extensible modeling language that captures the workload of different "business-logic" actions, e.g., memory allocations and database operations. WML compliments CBML because the actions (operations) in CBML can be parameterized using the workloads specified in WML. When both modeling languages are used together to model the behavior of a component, developers can specify both the actions of the component and the type of workload created by these actions.

Sidebar 1: Input/Output Automata

Input/Output (I/O), developed by Lynch and Tuttle, is a labeled transition system model for components in asynchronous concurrent systems. The actions of I/O automata are classified as *input*, *output* and *internal* actions, where input actions are required to be always enabled. I/O automata also have "tasks". In a fair execution of an I/O automata model, all tasks are required to get turns infinitely often. The behavior of an I/O automata model is describable in terms of traces, or alternatively in terms of fair traces. Both types of behavior notions are compositional.

Below we discuss a method for defining a behavior and workload modeling language similar to CBML and WML in CUTS for an MDE tool. We then discuss how to integrate the standalone behavior and workload modeling languages into existing DSMLs that currently only capture structure. The goal is to create an integrated DSML that allows users to capture both structure and behavior of a system and its components, as opposed to using of using separate tools or languages. To help illustrate this method, we use an example behavioral specification of the *Planner-1* component from the SLICE scenario. Table 1 highlights the behavioral specification of the Planner-1 component.

Planner-1	
Workload performed every second	Publish command of size 24 bytes
Workload performed after receipt of a track event	Allocate 30 KB; 55 dbase ops; 45 CPU ops; publish assessment of size 132 bytes; dealloc 30 KB

Table 1. Behavioral Specification for the Planner-1 Component in the SLICE Scenario

As shown in Table 1, *Planner-1* has two primary behavioral specifications. The first behavior is "operations performed every second," which sends a command event to both the sensor components. The second behavioral specification is workload performed after a track event is received from a sensor component. Upon receipt of a track event, the *Planner-1* component will complete a series of operations, and then transmit an *assessment* event to the Planner-2 component. For more information on the behavior of this and other components in the SLICE scenario see (Slaby, 2005). The remainder of the section focuses on defining a behavior and workload modeling language, and integrating it with existing structural DSML while using the Planner-1 component as a running example.

• ***Defining the behavioral modeling language.*** One goal of a behavioral DSML like CBML is to provide users with the necessary elements to capture the behavior of a component that is well-defined, but does not require any expertise in understanding mathematical formalisms. The DSML should, therefore, contain elements that are familiar to modelers, yet hide the complexity of the underlying formalism used to define the language. For example, I/O automata use action-to-state sequences to define behavior where the connection between an action and state must be an *effect* and the connection between a state and action must be a *transition*. Likewise, *preconditions* are associated with transitions and *postconditions* are associated with effects. DSMLs can help to shield modelers from low-level details of I/O automata.

Users of behavioral DSMLs are often less interested in the underlying semantics of a formal language than they are with using it effectively. DSML developers, therefore, must provide the necessary elements that represent the underlying formalism for behavior at a higher-level of abstraction, which usually entails capturing the minimal number of elements that allow the formalism to retain its semantics. For example, Figure 6 illustrates the elements in CBML that express the formal semantics of I/O automata.

Figure 6. Main Modeling Elements of CBML

In CBML, each element shown in Figure 6 corresponds to an I/O automata element. The *Input Action* element corresponds to input, which signifies the start of a behavioral specification. The *State* element represents a state in I/O automata, which must occur between two consecutive actions. The *Action* element corresponds to internal actions in I/O automata. *Output Action* element corresponds to output actions in I/O automata, which signify sending an event to trigger the start of another behavioral specification. The *Variable* element corresponds to variables in I/O automata, which can be used in guarded transitions between actions and states. Each element also allows developers to define the behavior (minus its workload) of a component without having prior knowledge of I/O automata semantics. Figure 7 shows a model using the elements shown in Figure 6 to define the behavior of the *Planner-1* component from the SLICE scenario in the case study. This behavior includes the input actions that start the behavior, action to state sequences, output actions, and pre-/post-conditions (not pictured).

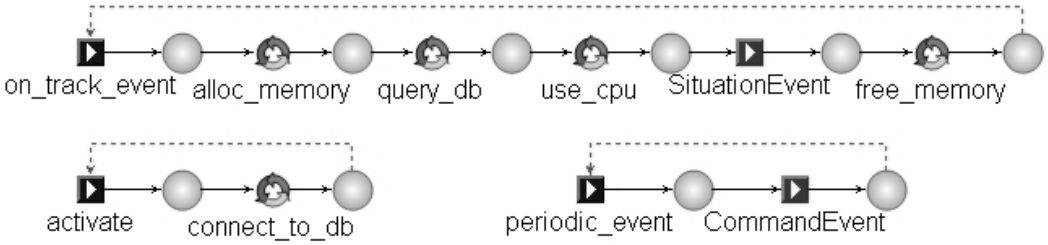

Figure 7. Example Behavioral Model using CBML

As shown in Figure 7, each behavioral specification begins with the input action model element. Figure 7, therefore, contains three separate, and independent, behaviors in this one model:

- The *track_event* signifies the actions to perform once a track event is received on the input port of the component as specified in Table 1.
- The *periodic_event* corresponds to the actions executed on a periodic basis while the component is active as specified in Table 1.
- The *activate* behavioral specification dictates the actions to perform when the component is being activated, such as establishing a persistent connection to the target database.

In some cases, the behavioral language may contain semantics that the average user may not understand, or can be simplified. For example, in I/O automata an action element must *always* be followed by a state element. Likewise in Petri nets (Peterson, 1977), a state element must *always* be connected to a transition element. In either case, DSML developers should provide modelers with the necessary tools to address this complexity. Most domain-specific metamodeling tools allow developers to create add-ons that can simply the modeling process. For example, CBML contains a plug-in that will auto-connect a new action added to the model to the previous state using the correct connection type, auto-generate a new state, connect the new state to the previously added action, and set the new state as the previous state. This plug-in helps simplify the modeling effort because users need not worry about the underlying semantics of I/O automata represented in CBML.

• ***Defining the workload modeling language.*** When designing DSMLs for capturing component behavior, we decoupled the workload specification from the behavioral specification for several reasons. First, this decoupling allowed both languages to evolve independently of each other as long as there is a common element that bridges between the two languages. Second, this decoupling allows the behavioral modeling language to interoperate with other workload modeling languages, such as framework-specific modeling languages (Antkiewicz, 2006a, 2006b), as long as they use the same bridging element defined in the behavioral modeling language.

When we defined WML, we extended the action and variable elements in CBML to create an object-oriented workload modeling language. Since our focus is SOA-based systems, we wanted to model the same development paradigm (i.e., object-oriented programming) used by SOA-based systems. Figure 8 shows the high-level overview of WML.

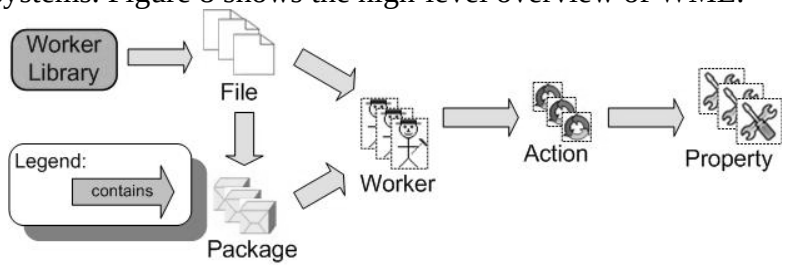

Figure 8. High-level Overview of WML Metamodel Structure

The modeling semantics in WML represent programming semantics similar to creating a shared library, i.e., .so files on UNIX and .dll files on Windows. The top-most element of the model is the *worker library*. The worker library is a shared library assembled from a collection of files (e.g., .cpp and .h files). Each file can contain one or more *worker* elements, which represent workload generators that can be used in an emulated environment. Each *worker* contains one or more *action* elements that represent the type of operations it can perform. Action elements can contain multiple *property* elements that represent parameters for that particular action. In CBML, we defined workers and actions to have the same modeling semantics as *variables* and *actions* in CBML, respectively. Since the *workers* and *actions* use the CBML bridging elements, modelers can then use WML in their existing behavioral models to give realistic workload parameters to the arbitrary actions. Figure 9 shows the CBML model in Figure 7 integrated with WML elements from Figure 8.

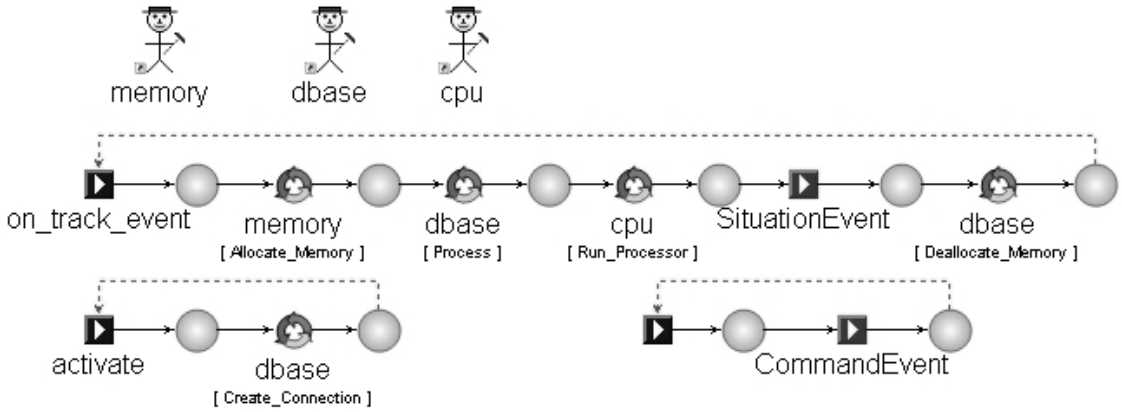

Figure 9. Integration of WML with CBML

As illustrated in Figure 9, the persons in the image represent worker elements illustrated in Figure 8. Each of the circular arrows is the actions of a respective worker (as illustrated in Figure 8). Unlike the actions in Figure 7, these actions are model instances of a preexisting action contained in a WML worker element. Since the actions in a WML and the actions in CBML have the same modeling semantics (i.e., act as bridging elements), it is possible it use variants of WML actions in CBML models, as illustrated in Figure 9. The remaining elements in Figure 9 are the same as the elements in Figure 7.

Regardless of whether WML is used to model the workload of a component, the point of decoupling the workload specification from the behavioral specification is to provide greater flexibility and extensibility. As illustrated in Figure 9, it is possible to retain the original behavioral model for a component, but interchange its "actions" as needed. If we chose to move to a different workload language or support multiple workload languages, therefore, the decoupled design of CBML and WML make it easy to integrate with other modeling languages.

• ***Integrating behavioral languages with existing structural languages.*** Earlier we focused on capturing component behavior using stand-alone DSMLs, namely CBML and WML. To leverage the power of the behavioral and workload DSMLs, however, they need to be integrated with existing structural modeling languages, such as PICML or J2EEML. These DSMLs provide developers with the necessary tools to generate metadata based on structural aspects, but provide no support for modeling behavioral aspects.

The structural DSMLs for SOA-based systems usually capture a component's interfaces and attributes, which can be viewed as the beginning of a behavioral specification. Likewise, a behavioral DSML usually has an element for specifying the initial action of its specification. It is, therefore, possible to extend existing structural languages with new behavioral languages by defining a connection (or bridge) between the stating actions of both the structural and behavioral aspects, as shown in Figure 10.

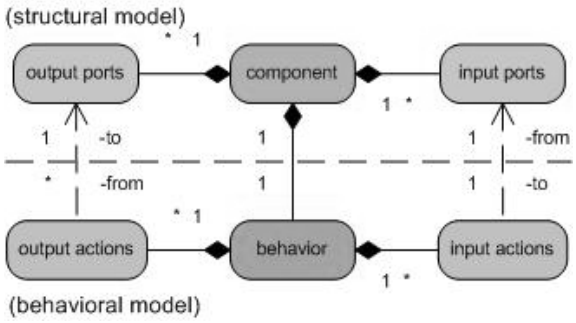

Figure 10. Conceptual Model of Integrating Behavioral Models with Structural Models

As shown in Figure 10, the upper portion shows the input and output elements of a typical structural DSML and the lower portion shows the typical input and output (I/O) elements of a behavioral DSML. The behavioral DSML contains I/O action elements that can be mapped to I/O ports, respectively, in the structural DSML. When we realize this mapping of elements from the behavioral DSML to elements in the structural DSML when integrating CBML and WML with PICML, we can create models similar to the one illustrated in Figure 11.

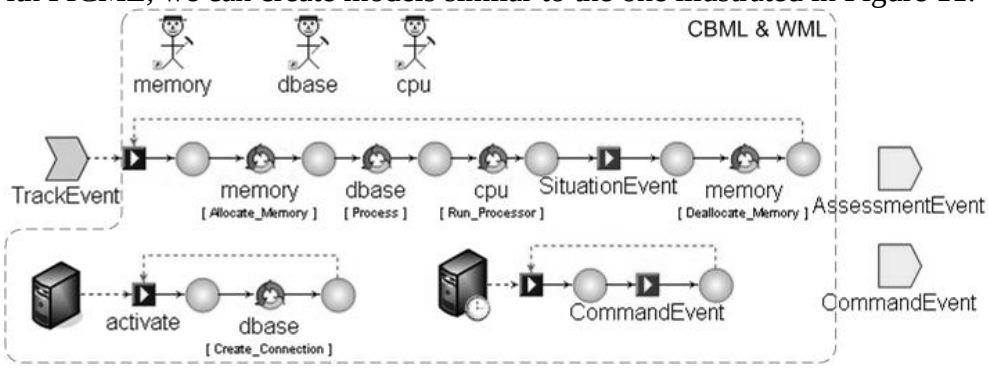

Figure 11. Integrated Model of CBML and WML with PICML

As shown in Figure 11, the elements inside the *CBML & WML* box highlight the same CBML and WML model for the *Planner-1* component in Figure 9. Likewise, the elements outside the box are PICML elements that define the structure of the *Planner-1* component. The entire image is the result of integrating CBML and WML into PICML, and capturing behavior and structure for the *Planner-1* component using a single DSML. Now that we are able to model a component's structure and behavior, we can now define model interpreters that will parse the models and generate the necessary output so that we can emulate the components' behavior, as discussed next.

Capturing Application Behavior for Emulation

Context. Next-generation SEM tools support emulation, i.e., running a variant of the system under development on the target platform. The target platform can either be a testbed that contains replicas of the real infrastructure's software and hardware or it can be the *real* infrastructure itself. In either case, next-generation SEM tools allow developers and testers to run emulations that leverage their target platform.

Problem → Determining how to emulate application behavior. When developing a SEM tool, an important design decision is determining how to emulate the system being developed on the target platform. For example, should the emulated system consist of (1) XML-based configuration files that can be interpreted and executed or (2) implementation code written in third-generation languages that is generated and compiled? Should the results of performance metrics be (1) written to a file or (2) gathered and transmitted to a central location, e.g., a database? The answers to these questions are important because they affect the design and usability of SEM tools.

Solution → Choose methods of emulation that offers flexibility, but meet application needs. When choosing a method of emulation, it is important to evaluate its impact on the usability and flexibility of a SEM tool. For example, using a XML-based implementation offers high configurability, but has run-time performance trade-offs. Likewise, collecting and transmitting data to a central location during the experiment adds more network traffic.

Applying the solution to CUTS and SLICE. When we designed CUTS and applied it to the SLICE scenario we evaluated the following features of a SEM tools design.

• ***Component instrumentation methods.*** There are two design choices to select from when trying to emulate a system. The first design choice is to use *non-replaceable* components that mimic the behavior and workload of its realistic counterpart, but do not have the expected interface as the actual components. The motivation for using non-replaceable components is they are straightforward to implement and generate from a model since they need not conform to any specification (i.e., use the correct interfaces or calling conventions), unlike real components.

Non-replaceable components typically use text-based configuration files (Slaby, 2006). For example, an XML metadata file can contain the behavior and workload characteristics of a particular component. When the system is deployed, a generic component (or object) will read the appropriate XML metadata file and configure itself accordingly. Although this design choice offers great flexibility, it incurs the overhead of interpreting the contents of the configuration file. As the real components are developed, moreover, they cannot be integrated into the emulation environment due to the disconnect between the emulated component and the real component.

The other design choice is *replaceable* components, which have the same interfaces and attributes like their real components and can be swapped out for real components once their development is complete. This design allows continuous system integration from design-time to production-time (Hill, 2006). It is common for replaceable components to consist of implementations generated directly from model, similar to an IDL compiler generating stubs and skeletons from an IDL file. Another advantage of this approach is that implementations can be generated on a per-component/model basis to achieve the most accurate emulation results possible. The downside of approach, however, is that it requires developers to recompile components when their behavioral model changes causing regeneration of the emulation code.

CUTS uses *replaceable* components that provide the same interfaces and attributes as their real counterparts. It processes the models specified by users and generates implementation code and project files needed to compile the complete system. In addition, CUTS implements a hybrid between purely replaceable and non-replaceable emulation components, which generates replaceable components that use text-based configuration files to determine their behavior. The advantage of this hybrid approach is that developers and testers only regenerate and recompile implementation code if the *structure* of the system changes. If the behavior of the system changes then only the text-based configuration file is regenerated.

• ***Benchmark methods.*** When emulating a system with replaceable or non-replaceable components, it is necessary to benchmark its performance by collecting performance metrics. There are two general methods for benchmarking a component: *intrusive* (Menascé, 2004) and *non-intrusive* (Mania, 2002; Parsons, 2006). Intrusive benchmarking requires developers to annotate their existing source code with new code that collects the necessary metrics. The advantage of this approach is that developers and testers can dictate which application-level metrics to collect. The disadvantage of this approach is that testers and developers must manage the metrics collection themselves, which may involve creating a benchmarking framework and/or learning how to interface with an existing one.

Non-intrusive benchmarking does not require any modifications to the existing source code. Non-intrusive benchmarking, in contrast, uses external or infrastructure-level tools to monitor different aspects of the system, such as disk and memory usage or arrival of an event. It is also common to use proxies (Parsons, 2006) that host the real component to capture performance metrics non-intrusively. The proxies resemble the real components and record performance metrics as events enter and leave the component to which it delegates to and from. If application-level metrics, such as invoking an operation that is internal to a component, are need, however, intrusive monitoring is necessary since non-intrusive techniques have no knowledge of application-level implementation.

CUTS provides both non-intrusive and intrusive monitoring. Non-intrusive monitoring is achieved using proxies that have the same interfaces and attributes as their hosted components. The proxy monitors all events that enter and leave the component to which it delegates to and from. If application-level metrics are needed, CUTS allows developers to log the metrics to a thread-specific logging record associated with the source event using simplified intrusive monitoring techniques.

• ***Data collection methods.*** Another design choice that influences system emulation is choosing the data collection method, which can either be *offline* or *online*. In offline collection, metrics are written to a file while the system is being emulated. After an experiment is complete the metric file is analyzed using an analysis tool. The advantage of this method is that experimenters can determine the format of the output file and what metrics are written to the file. The disadvantage is the metrics usually cannot be processed until the experiment is complete, which can pose a problem during long running experiments or when trying to monitor the progress of an experiment.

In online analysis, metrics are collected and transmitted via network to a host outside the experiment environment. The advantage of this approach is that it allows analysis of metrics in an environment that does not use the experiments resources, so the experiments will not skew the results while it is running. The disadvantage is the difficulty of devising a strategy for efficiently collecting metrics in a distributed environment and submitting them to a central location without negatively impacting the running experiment, especially in an experiment with lots of network traffic.

CUTS uses an online distributed data collection technique that collects metrics in three stages. In stage 1, each port of a component records its performance metrics, e.g., number of events received, the max/min transmission and processing time, and any application-level metrics. In stage 2, an agent within the proxy collects the data from each port at a user-specified interval and resets the each port's records. In stage 3, the agent transmits the gathered data to a central location called the *BenchmarkDataCollector*, which writes the collected data to a file or database. The BenchmarkDataCollector also allow external services to query it directly to evaluate online performance. For example, a component could query the BenchmarkDataCollector for the latest execution times of each component to determine how to adjust their priority level.

Analysis of Performance Metrics for Informative Feedback of Emulation Results
Context. Analysis of performance metrics is a key part of SEM tools. Whether metrics are collected *online* or *offline*, it is necessary to understand what the collected metrics mean so that design flaws can be located and rectified. If metrics are collected *online* the analysis tool needs to provide testers with the ability to view partial results of the collected metrics until the experiment is complete. Once the experiment is complete, the tool will then provide an overall analysis of the experiment. If metrics are collected *offline*, then the tool needs to support joining multiple data collection files to provide a synopsis of the experiment.

Problem → Providing meaningful analysis of performance metrics. The power of an analysis tool depends on how the metrics are presented testers. Visual analysis generally works best when viewing large amounts of data since visualization helps partition (or cluster) the data so that it is intuitive to readers. Visualization also includes using charts (e.g., pie charts, bar charts, and timelines) that allow developers and testers to highlight points of interest, obtain a general or detailed view of metrics (Herman, 2000), or save and correlate metrics with multiple experiments (Hauswirth, 2005). Achieving this goal, however, is hard because not only must performance metrics be analyzed, but visual aids must be created to present the analyzed data.

Solution → Leverage existing visualization packages. Many packages—both open-source and commercial—can be used to help analyze collected metrics. With the advent of service-oriented architectures, many analysis tools, such as WebCharts (www.carlosag.net/Tools/WebChart), WebCharts3D (www.gpoint.com), and Dundas Charts (www.dundas.com), provide testers with the ability to create powerful visual aids that can integrate with web applications and viewed online for any place that is accessible to the internet. Therefore, it is possible to provide analytical support with little effort from developers of data analyzers.

Applying the solution to CUTS and SLICE. In CUTS, analysis is done using at tool called the Benchmark Manager Web (BMW) Utility. The BMW is a Microsoft .NET web application that uses a third-party web charting tool to present metrics collected during an experiment. For each experiment, the BMW provides testers with an overall synopsis of the current tests, as well as detailed information (e.g., deployment information) for each component or timeline of application-level performance metrics, as shown in two example tables from the SLICE scenario in Figure 12.

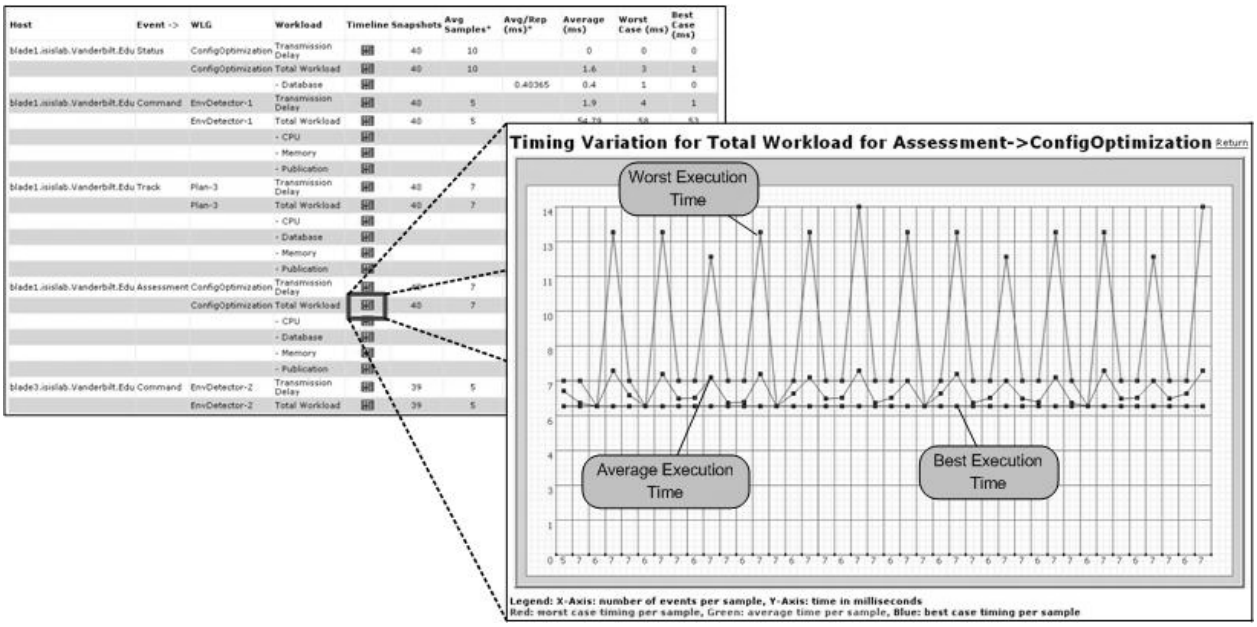

Figure 12. General (left) and Detailed (right) View of Performance Metrics Using the BMW

More specifically, the general view in the left-hand portion of Figure 12 shows each of the deployed components, where each component is deployed, the number of events transmitted/received by a component, and the best-, average-, and worst-case service time of an event for each component. The detailed view in the right-hand portion of Figure 12 gives a chronological timeline of the best-, average-, and worst-case execution times for a particular event in a component.

Figure 12 shows that the timing of the total workload exhibits semi-periodic behavior and much jitter. These results occur because the component from Figure 12 is deployed on the same host as other components and thus competes for resources. Likewise, as events travel from component to component in the system, the location of each component (i.e., which host contains the component) affects timing synchronization between nodes and resources. The semi-periodic behavior in Figure 12 is, therefore, illustrative of how the placement of a system component affects other component's performance in relation to the structure of the overall system. Although we do not provide this form of analysis, it is possible to use correlation analysis (see Evaluation Techniques for Component Architectures section) or *main effects screening* (Yilmaz, 2005) techniques to analysis collected performance metrics for these properties.

In addition to providing a detailed view of an event in a component, the BMW provides a detailed view of transmitting an event through a serious of component, which we call a *critical path*. Figure 13 shows a detailed view of analyzing a critical path using the BMW for the SLICE scenario. The upper graph in this figure depicts the average case time it took to transmit an event from main sensor to the main effect of the SLICE scenario as explained in the case study. The lower graph in Figure 13 depicts the worst-case time for the same critical path. In either graph, there are two separate bar graphs. The upper bar graph is the actual time measurements collected by the non-intrusive benchmarking methods for a single event as it passes through the each component in the critical path, as show by each block. The lower bar graph illustrates the deadline for the critical path.

If the deadline for a critical path is achieved, i.e., the event passes through all the respective component before missing its deadline, the lower bar graph will have a green strip to indicate successful completion of the critical path within its deadline, which is illustrated as "headroom" in the average-case graph. If the deadline is missed, the lower bar graph will have a red strip to indicate failure to complete deadline within specified time, which is shown as "overrun" in the worst-case graph. There graphs created by the BMW allows users to closely monitor the performance of events in the target system. Moreover, it also helps users pinpoint bottleneck components in the system and improve deployments for subsequent experiments.

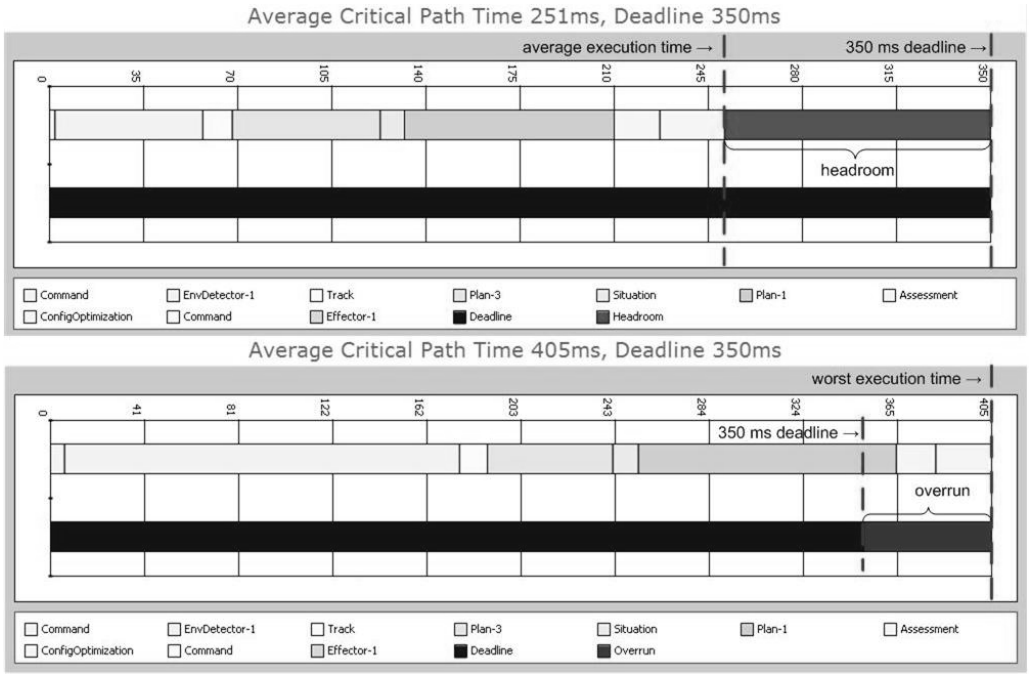

Figure 13. Detailed View of Analyzing a Critical Path Using the BMW

The BMW is also a web service, which means it can process SOAP requests. Support for SOAP allows CUTS to query the state of a remote experiment using any programming language that can send and receive SOAP messages. More importantly, the BMW web service can also control the state of an experiment, including starting, stopping and pausing an ex-

periment. The BMW, therefore, allows experimenters to manage experiments remotely for any location accessible via the Internet.

EXPERIMENTAL RESULTS

To illustrate the benefits of next-generation SEM tools, i.e., those integrated with MDE technologies, we present the results of an experiment that applies CUTS to the SLICE scenario introduced in the case study. Our experiment explores the following two hypotheses:

1. The components in the SLICE scenario produce too much workload to all be deployed on the same host and still meet the 350ms deadline using the expected software and hardware configuration of the target environment.
2. It is possible to use CUTS to locate a collection of deployments (i.e., more than one) that will allow the critical path of the SLICE scenario to achieve its 350 ms deadline while meeting the deployment requirements specified in the case study.

This section discusses our process of using CUTS to evaluate the two hypotheses stated above. It discusses using CBML and WML in CUTS to capture the behavior and workload of the components in the SLICE scenario. The CBML and WML models are then used to generate source code for replaceable emulation components. Lastly, we present the results of eleven different tests conducted in the target environments using the emulated components to analyze system performance based on the stated hypotheses. We conducted eleven tests because, after eleven different tests, we were able to answer the two hypotheses. The eleven tests, however, are *not completely exhaustive* of all possible deployment and configurations models of the SLICE scenario.

Specifying the Behavior of the SLICE Components

The first step in applying CUTS, or any other SEM tool, is to specify the behavior of each component.[1] Developers are typically given a high-level specification, which is usually a text-based document, of each component's expected behavior, or role in the system. This information must then be interpreted into a behavioral model that represents, as best as possible, the expected behavior of the system and its components. For example, Table 1 contains the behavioral specification for the *Planner-1*component of the SLICE scenario.

When we convert the behavioral specification in Table 1 into a model using CBML and WML, we get the model shown in Figure 11. Using same method for constructing *Planner-1*'s behavioral model, we constructed models for all the remaining components in the SLICE scenario. Afterwards, we generated components for emulation from the models. The generated components had the same interface and attributes as their real components, so that as the real components were developed, they could replace the faux components. Finally, we used other structural tools in CoSMIC to generate the configuration and deployment metadata descriptor using the faux components and deployed the emulated system via the same deployment tools used in the target environment.

Emulating Application Behavior to Evaluate End-to-End QoS

One of our goals in the case study was to evaluate the end-to-end QoS of the SLICE scenario during early stages of development, as opposed to system integration time, thereby making it easier to locate and rectify performance problems as early as possible. Moreover, we wanted

[1] We assumption the structure of the system and its components have already been modeled using MDE tools.

to locate a set of deployments, i.e., placement of components onto hosts, that will allow the critical path of the SLICE scenario to run in <= 350 ms (hypothesis 2). To avoid a single point of failure, the SLICE scenario also required the deployment of components in the critical path across multiple hosts, and the main sensor and effector had to be deployed on separate hosts. We, however, wanted to determine if it was possible to meet the 350 ms critical path deadline when all the components were deployed on a single node (hypothesis 1). We, therefore, ran a series of experiments using CUTS to locate a set of deployments that satisfied the design and performance requirements.

Each host in the experiment was an IBM Blade Type L20, dual-CPU 2.8 GHz processor with 1 GB RAM with the characteristics listed in Table 2. The middleware was version 0.4.7 of CIAO/DAnCE (Wang, 2002; Deng, 2005) and the MDE tools were version 0.4.6 of CoSMIC (Gokhale, 2006), which is the target middleware and MDE tool for the SLICE scenario.

Host	**Operating System**	**Database**
1	Fedora Core3	YES
2, 3, BDC	Fedora Core3	NO
BMW	Windows XP	YES

Table 2. System Characteristics for Each Host in the SLICE Experiment

Table 3 presents the results of eleven different experiments of the SLICE scenario using CUTS. Each experiment was run for 10 minutes to allow the collected performance metrics to stabilize. We were able to verify performance metric stabilization by using the online metrics analysis capabilities of the BMW. We ran eleven tests because after the final test we had enough data to answer two hypotheses about the SLICE scenario. Moreover, we were able to show that CUTS could be used to analyze performance during the early stages of development to address questions about end-to-end system performance earlier in the design phase. Such questions included locating a collection of deployments that met the 350 ms critical path deadline or determining if the SLICE scenario produced too much workload for a single host based on the current software/hardware configuration of the target environment.

SLICE CoWorkEr Legend for Test Table			
Symbol	**CoWorkEr**	**Symbol**	**CoWorkEr**
A	Sensor-1 *	E	Config-Op *
B	Sensor-2	F	Error-Recovery
C	Planner-2 *	G	Effector-1 *
D	Planner-1 *	H	Effector-2
** represents CoWorkEr in the critical path*			

Test	**Deployment Strategy**			**Critical Path Execution (avg./worse) (ms)**
	Host 1	**Host 2**	**Host 3**	
1	C,D,E,F	A,B	G,H	411 / 1,028
2	A,B,C,D	F	E,G,H	420 / 1,094
3	A,B,C,D,E	F	G,H	416,/ 1,085
4	A,B,C,D,E,F,G,H			463 / 1,247
5	A,B,C,D,E,G,H	F		467 / 1,219

6	A,C,D,E,G	F	B,H	323 / 844
7	A,G	C,D,E	B,F,H	363 / 887
8	D	A,B,C, F,G,H	E	405 / 975
9	A,D	C,E,G	B,F,H	235 / 387
10	A,D	E,G	B,C,F,H	251 / 395
11	A,D,E	C,G	B,F,H	221 / 343

Table 3: Results of SLICE Scenario for Different Deployment s

Analyzing the Performance Results of the SLICE Scenario

Table 2 presents the results of eleven different tests we conducted for the SLICE scenario using CUTS. Only three of the eleven tests (Tests 9, 10, and 11) had a deployment model where the critical path components deployed across multiple nodes and, when emulated by CUTS, completed their end-to-end execution in 350 ms. From Test 9, 10 and 11, we were able to start a collection of deployments for the SLICE scenario, which addressed hypothesis 2. Of those three tests, only two tests (Test 9 and 11) had a deployment model where the critical path was deployed on two separate nodes, and completed their end-to-end deadline in 350 ms. One experiment (test 6) completed the critical path in 350 ms, however, the critical path components were all deployed on a single host, e.g., host 1. Only one of the tests (test 11) completed the critical path within the worst-case execution time of 350 ms. Although we did not exhaust all possible deployment strategies with these tests, we learned that only 27% (3 out of 11) of the current test passed on their planned infrastructure while meeting the deployment requirements and Test 11 yielded the best performance.

• ***Measuring the limitations of single host deployment.*** The deployment constraints of the SLICE scenario described in the case study requires all the components in the critical path to be deployed across multiple nodes. In addition, the main sensor (Sensor-1) and the main effector (Effector-1) must be deployed on separate nodes to avoid a single point of failure. If the components are deployed across multiple hosts it is possible to simplify system recovery though redeployment (Shankaran, 2005). Although the requirements specify that the critical path components must be deployed across multiple nodes, we wanted to determine if it was possible to deploy all the components (in the entire application and only in the critical path) on a single node and still meet the 350 ms deadline to verify hypothesis 1.

After running Test 4 and 5, we realized that it would be impossible to meet the 350 ms deadline when all the components are deployed on the same host (i.e., the SLICE scenario produced more workload than a single host could handle as conjectured in hypothesis 1) for our current hardware and software configuration. When we deployed the components in the critical path on one host and the remaining components to a separate host (Test 6), we met the 350 ms critical path deadline for our current hardware and software configuration. We were, therefore, able to determine the possibility of achieving the 350 ms deadline when all components are deployed on a single host, and when only the components in the critical path are deployed on a same host.

• ***In-depth analysis and understanding of results for hypothesis 2.*** After running Test 1 through 8, only one test (Test 6) met the 350 ms end-to-end deadline. Moreover, seven of the

tests had faults in their deployment specification, e.g., the combination of particular components on a host generated more workload than the host could handle to meet 350 ms critical path deadline. To pinpoint the bottlenecks, we used CUTS graphical analysis features to investigate why these deployment strategies did not meet their QoS requirements.

Host	Event ->	WLG	Workload	Timeline	Snapshots	Avg Samples*	Avg/Rep (ms)*	Average (ms)	Worst Case (ms)	Best Case (ms)
blade5.isislab.Vanderbilt.Edu	Command	EnvDetector-1	Transmission Delay		39	5		6.19	7	6
		EnvDetector-1	Total Workload		39	5		169.6	522	54
			- CPU				2.20859	99.39	208	16
			- Memory				0.00727	0.51	1	0
			- Publication				1.40206	1.4	20	1

Figure 14. Snapshot of Timing Data for Sensor-1 in Test 8 obtained from the BMW Test Results Page

Figure 14 and 15 illustrate the results provided via the BMW for Test 8, which measures the behavior when two components in the critical path handling the most workload are deployed on their own node. Figure 14 shows the time to transmit a message to the *Sensor-1* (EnvDectector-1) and how long it took to complete each type of workload (e.g., CPU, database, or memory) for the *Sensor-1*. Likewise, Figure 15 shows the average execution time of an event through each component in the critical path of the SLICE scenario. We observed that *Sensor-1* took 169.6 ms to process its workload after receiving a *command* event from *Planner-1*, *Planner-1* took 54.0 ms to perform its workload after receiving a *track* event from *Sensor-1* or *Sensor-2*; and *Planner-2* took 110.6 ms to perform its workload after receiving a *command* event from *Planner-1*.

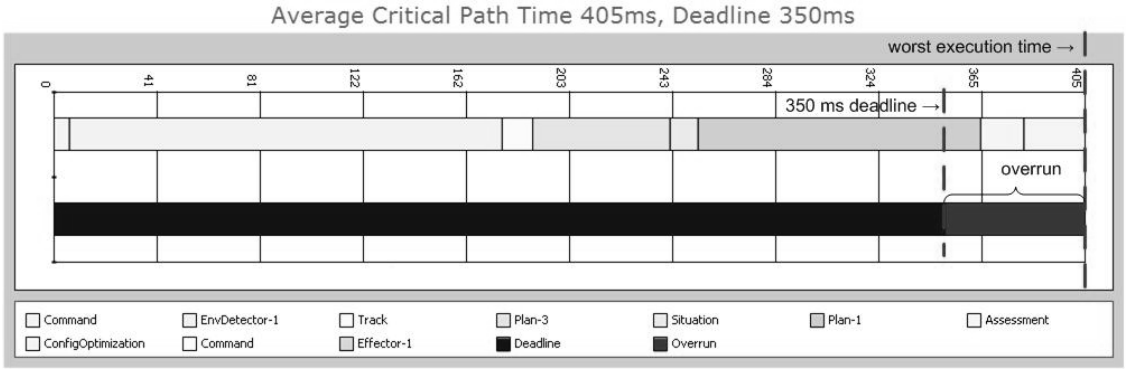

Figure 15. Snapshot of the Critical Path Timing Data for Test 8 from the BMW Analysis Page

For Test 8, *Sensor-1* and *Planner-2* have the longest completion times. Based on the quantitative analysis provided by CUTS, we realized that the *Sensor-1* and *Planner-2* CoWorkEr components had a heavier workload than expected, and must be deployed on separate nodes. We, therefore, created a new deployment model that placed *Sensor-1* and *Planner-2* CoWorkErs on different hosts, which lead to the deployment models used in Test 9, 10 and 11, all of which met the 350 ms end-to-end deadline. Of these three tests, Test 11 (shown in Figure 16) was the only test to have a worst-case execution time that met the 350 ms deadline. These deployment strategies also met the deployment requirements of placing *Sensor-1* and *Effector-1* on different nodes. We were, therefore, able to answer our second hypothesis,

which was using CUTS to locate a collection of deployments that meet the 350 ms critical path deadline

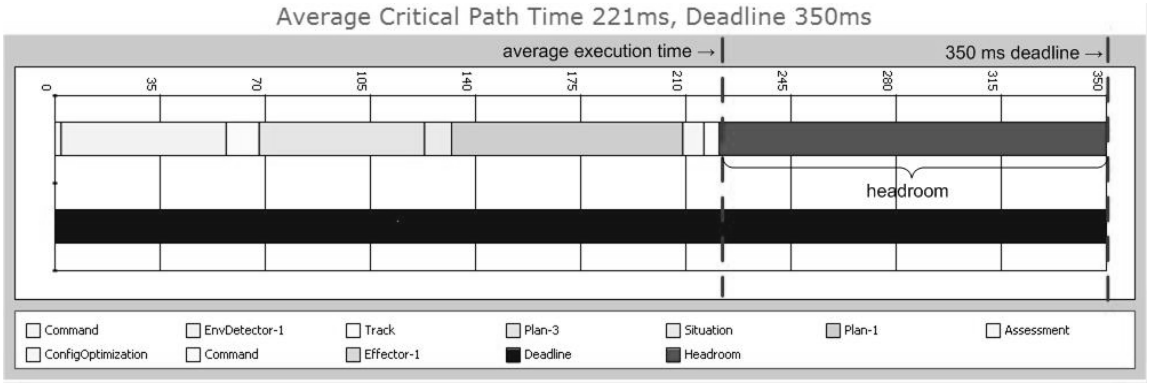

Figure 16. Snapshot of the Critical Path Timing Data for Test 11 from the BMW Analysis Page

Summary of the Experimental Results

Our experience applying CUTS to the SLICE scenario showed how next-generation SEM tools can help decrease time spent resolving integration problems early in the development lifecycle. Instead of waiting until full system integration, CUTS allowed us to test multiple deployments of the SLICE scenario in the target environment using emulated application components. The results of our eleven tests for the SLICE scenario, which was not exhaustive of all possible tests, allowed us to create a collection of deployments that meet the 350 ms critical path deadline during the early stages of development. Moreover, we verified that the SLICE scenario produced more workload than a single host in our target environment could handle.

As the real components for the SLICE scenario are completed, we can integrate them into the emulation environment to achieve more realistic results since CUTS implements replaceable emulation components. This incremental replacement process allows developers to perform continuous system integration from design-time (i.e., early stages of development) to integration-time (i.e., final stages of development). As a result, the collection of deployments that meet the 350 ms deadline from the continuous system integration using CUTS will be the same deployments system engineers use when the system is in production.

FUTURE TRENDS

This section discusses emerging and future technological trends associated with developing enterprise DRE systems using SEM tools, with an emphasis on J2EE and Microsoft .NET technologies since many enterprise business applications are developed using J2EE and Microsoft .NET, so this section balances out the earlier focus on the CORBA Component Model.

Increased use of MDE technologies for code generation. MDE technologies are increasingly being applied to address many of the challenges and complexities of developing enterprise DRE system (AndroMDA, 2007; Margaria, 2004; Smith, 2006; Task, 2006). As a re-

sult, there is a shift from the traditional development paradigm—where developers handcraft most artifacts, such as source code and configuration files—to an MDE approach where developers use domain-specific modeling languages to develop enterprise DRE systems. A key advantage of an MDE approach is that it shields application developers from many error-prone and tedious tasks, such as manually scripting dense and complex XML files or writing redundant source code that can instead be generated from models.

For example, framework-specific modeling languages (FSMLs) (Antkiewicz, 2006a, 2006b), which capture the API of a framework and ensure proper usage of its building blocks, are increasingly being integrated into existing MDE tools, such as the Workbench Part Interaction (WPI) FSML prototype (Antkiewicz, 2006c) or Pattern-Oriented Software Architecture Modeling Language (POSAML) (Kaul, 2007). An FSML can be used to generate valid source code directly from the models. Likewise, as the underlying framework changes between versions, the FSML can capture these revisions, such as deprecated methods, new methods of an object, or modified parameters of an pre-existing method. Moreover, the FSML will enforce the changes, e.g., make modelers aware of revisions in the target framework so they can correct their models accordingly via round-trip engineering.

Increased use of continuous integration servers to improve software quality. *Continuous integration* (Fowler, 2006) environments are a form of extreme programming (XP) (Beck, 2000) where integration is accomplished by server daemons using serialized build processes. Continuous integration environments such as Build Forge (www.buildforge.com), CruiseControl (cruisecontrol.sourceforge.net), and DART (public.kitware.com/Dart) continuously exercise the complete build cycle of a product to ensure that software is of the highest quality by:

1. Performing automated builds of the system upon source code check in or successful execution and evaluation of prior events;
2. Executing suites of unit tests to verify basic system functionality;
3. Evaluating source code to ensure it meets coding standards and best practices; and
4. Executing code coverage analysis.

By utilizing continuous integration servers, developers can improve software quality because much of the manual labor required to stay abreast with large-scale software development will be handled autonomously. Moreover, instead of manually managing the software development process, more time and effort can be spent responding to development concerns (and problems) identified by continuous integration servers.

Continuous integration servers can alleviate many manually tasks introduced during the software development process, such as monitoring source code repositories to ensure the latest version of software builds successfully, or running unit tests to ensure proper functionality. Likewise, system execution modeling (SEM) tool suites provide developers with tools for testing applications in realistic environments (i.e., on the target architecture) using realistic workloads. Because system execution modeling tools offer features that continuous integration servers lack, such as realistic testing environments, and continuous integration servers offer services that SEM tool suites lack, such as management of large numbers of tests, marrying SEM tools suites with continuous integration servers will also help improve software

quality. By capitalizing from the integration of both SEM tool suites and continuous integration servers, developers will also be able to improve assurance of QoS.

J2EE and Microsoft .NET business applications. SOA technologies such as J2EE and Microsoft .NET are commonly used by enterprises to build business applications (IDC, 2005). As a result, SEM tools are emerging to support such applications through the concept of *process modeling* (Curtis, 1992), which is similar to behavioral modeling in SEM tools, but can operate at a higher level of abstraction. In process modeling, developers use next-generation SEM tools to capture the *workflow* (or "business-logic") of their target application. These models can then be used to run simulations/emulations to verify the application's correctness, generate a prototype of the application, or generate a production application that is integrated with the target SOA technology.

Windows Workflow Foundation (WinFX Workflow) (Box, 2006), which is part of the Microsoft .NET 3.0 framework, allows developers to model the business processes of their enterprise business applications. WinFX Workflow contains the following parts related to process modeling and generation of business applications:

- **Activity model** that allows developers to capture the different actions, or work working units, of the business applications, e.g., its operations and workloads.
- **Workflow designer** that allows developers to sequence activity models to define the behavior of the application's business logic.
- **Workflow runtime** that allows developers to execute the workflows of the business application created in the workflow designer in an emulated environment or target environment.

The Java Workflow Tooling (JWT) (Dutto, 2007) is another MDE technology for process modeling. JWT targets J2EE business applications, and is still under development. Similar to WinFX Workflow, JWT offers developers of J2EE applications the ability to capture the workflow of their business applications, and execute the workflows in the target environment. The JWT is comprised of the following parts:

- **Workflow Editor (WE)** that is a visual tool for creating, managing, and reviewing process definitions, i.e., their business logic.
- **Workflow engine Administration and Monitoring tool (WAM)** that is used to execute, monitor, and analyze workflows of business applications created using the workflow editor.

The Business Process Modeling Notation (BPMN) (BPMN, 2005) is a standard developed by Business Process Management Initiative (BPMI) that allows developers draw business processes in the form of workflows. BPMN is comprised of the following parts for graphing workflows:

- **Flow objects** (e.g., event, activity, and gateway) that determine how the business process behaves, or flows.

- **Connecting objects** (e.g., sequence flow, messaging flow, and association) that are used to connect one or more flow objects. This allows developers to sequence the flow objects to create workflows, or process models.
- **Swimlanes** (e.g., pool and lane) that are used to organize workflows into categories and groups within the categories, respectively.
- **Artifacts** (e.g., data objects, group, annotation) that allow developers to add information to the model that does not affect workflow, and makes it more comprehendible.

Similar to WinFX and JWT, BPMN can be transformed into an execution language called Business Process Execution Language (BPEL) (White, March 2005).

The major difference between BPMN, JWT, and WinFX Workflow is their target SOA technology. WinFX is designed for Microsoft .NET applications, where as the JWT is designed for J2EE application. Likewise, BPMN is a technology independent graphical language for graphing business processes as workflows. Although each technology has their differences, it is clear that using MDE technologies to model business processes, i.e., model their behavior, perform analysis checks, and generate the target application from the model for emulation, or production, is an important trend in future MDE technologies.

FUTURE RESEARCH DIRECTIONS

Many existing enterprise applications have been developed using traditional development techniques, i.e., applications were build directly on top of operating systems and networking protocols. Consequently, many enterprises developed their own (distributed) middleware, which has become deeply embedded into—and the foundation of—many subsequent applications. Enterprises applications are now evaluating the benefits of service-oriented architectures (SOAs), such as CORBA Component Model, Microsoft .NET, and J2EE, and are increasingly migrating to this programming paradigm.

Although SOAs provide many benefits, such as encapsulation of business-logic in components for reuse, existing performance analysis techniques still rely on low-level traditional profiling techniques (Waddington, 2007) because of the reliability and maturity of existing tools and techniques. For example, DTrace (Cantrill, 2004) is a powerful profiling tool distributed with the Solaris operating system that uses low-level tracing techniques (*e.g.*, tracing kernel- and user-level functions/variables) to locate and resolve performance issues. The Java virtual machine (JVM) profiler interface (Binder, 2005) is another example that allows developers to implement third-party profiler applications that interact with the JVM. Similar to DTrace, the JVM profiler interfaces allow profiler applications to monitor virtual machine- and user-level events while Java applications are executing.

Although tools such as DTrace and the JVM profiler interface are beneficial, SOA-based development techniques operate at higher level of abstraction than existing profiling tools support effectively. For example, a single node (such as a server) could host multiple components of the same type, i.e., each component is an instance of the same component type. Consequently, low-level profiling tools will not be able to distinguish between performance issues related to each component. Next-generation system profiling to similar to CUTS can

distinsuigh between performance issues related to each component, though they cannot provide the same low-level details as traditional profiling tools.

Future research is, therefore, needed to understand how to integrate low-level profiling tools with next-generation tools that operate at the component level. Although SOAs manage low-level implementation details, such as interaction with operating system APIs and networking protocols, the ability to profile applications using low-level tools can provide insight as to how to configure the SOA middleware properly. Moreover, profiling application behavior at the component-level will allow developers to understand how their components interact with the underlying SOA middleware (and operating system) so developers can provide the best implementation that are most beneficial to their applications.

CONCLUDING REMARKS

This chapter motivated the need for combining system execution modeling (SEM) tools with model-driven engineering (MDE) technologies to address the development and integration challenges of service-oriented architecture (SOA)-based enterprise distributed real-time and embedded (DRE) systems. To meet this need, we discussed the necessary ingredients (i.e., behavioral and structural modeling languages, emulation techniques, and analysis methods) and describe key challenges to overcome to guide in developing next-generation SEM tools in general while using our next-generation MDE-based SEM tool called CUTS as an example. We also showed how CUTS could be applied to the SLICE case study from the domain of shipboard computing to address integration challenges during early stages of development.

The following summarizes the benefits of applying CUTS to evaluate the QoS of enterprise DRE systems based on our experience thus far:

- **Early integration testing.** CUTS allowed us to emulate system components using the target hardware and software infrastructure. More importantly, we were able to emulate the system at early stages of development instead of waiting until completely implementing the real components and trying to resolve all issues during integration phase. We had attempted this is previous stages of the MLRM project described in the case study, but were unsuccessful since we missed project deadlines and had an increase in project effort..

- **Extensive QoS testing.** CUTS allowed us to rapidly create and quantitatively evaluate a range of deployment plans to see how they impacted end-to-end QoS behavior. Much more time and effort would have been required if these tests were conducted manually, i.e.,without the visual SEM tool functionality and automation provided by CUTS and the underlying CIAO and DAnCE QoS-enabled middleware and CoSMIC MDE tools. More importantly, CUTS provided qualitative performance analysis to assist in locating deficiencies in current deployments so we can determine alternative deployments that meet end-to-end QoS requirements more effectively.

- **Continuous system integration testing.** The use of SEM tools enabled us to substitute real components for the emulated ones quickly, so we could incrementally evaluate QoS performance with more realistic workloads as knowledge of the application and system

infrastructure evolves. This emulation enabled us to benchmark the performance of the system to evaluate QoS continuously. Moreover, it helped reduce the amount of time and effort spent during system integration trying to resolve the challenges we had addressed since the early stages of development.

Although there are many benefits to using CUTS to evaluate QoS of enterprise DRE systems, we also discovered that the following work is needed to improve the evaluation of QoS in component-based enterprise DRE systems:

- **Functional testing for (in)correctness.** It is becoming common practice to unit-test the business-logic of a component throughout the development lifecycle using continuous integration tools (Fowler, 2006). This helps increase confidence that the underlying framework used in the component is functioning correctly. When the business-logic is encapsulated inside a component, functional testing usually does not occur until integration testing (Li, 2005). Future work, therefore, is need to allow CUTS to provide unit testing for systems at the component-level (e.g., verifying input/output values are translated correctly between the components interface and business-logic, or exceptions are interpreted and handled correctly at the component-level), while utilizing the QoS testing features already provided by CUTS.

- **Pluggable QoS analysis capabilities.** Currently, CUTS provides minimum QoS analysis, such as end-to-end execution timing analysis and worst-case scenario analysis. We are, however, learning that there are times when our analysis tools may not provide enough details in certain situations, such as correlation analysis across multiple tests based on performance metrics or analyzing proprietary systems and data. Future work is therefore needed to extend CUTS to support pluggable analysis tools (or objects) that can analyze collected metrics and collect metrics we do not collect (e.g., input/output parameter values or current state of a component) at run-time.

CUTS is currently being transitioned from the MLRM project to a production Naval shipbuilding program to assist system engineers and architects in evaluating QoS performance metrics of DRE systems. An open-source version of CUTS and the other MDE tools and QoS-enabled SOA-based middleware platforms described in this paper can be downloaded from www.dre.vanderbilt.edu/CUTS.

REFERENCES

AndroMDA (2007). team.andromda.org.

Antkiewicz , M. (2006a, September). Round-trip engineering of framework-based software using framework-specific modeling languages. In proceeding of *21st IEEE International Conference on Automated Software Engineering*, 323 – 326, Washington, DC, USA.

Antkiewicz, M. & Czarnecki, K. (2006b, October). Framework-specific modeling languages with round-trip engineering. In proceeding of *9th International Conference on Model Driven Engineering Languages and Systems*, 692 – 706, Genova, Italy.

Antkiewicz, M. & Czarnecki, K. (2006c, October). Round-trip engineering of eclipse plug-ins using eclipse workbench part interaction FSML. In *Proceedings of International*

Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA). Portland, OR.

Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A. & Schmidt, D. C. (2005). A platform-independent component modeling language for distributed real-time and embedded systems. In proceedings of the *11th Realtime Technology and Application Symposium* (pp. 190–199), San Francisco, CA.

Baude, F., Caromel, D., Huet, F., Mestre, L., & Vayssiere, J. (2002). Interactive and descriptor-based deployment of object-oriented grid applications. *Proceedings of the 11th International Symposium on High Performance Distributed Computing*, Edinburgh, UK.

Beck, K. (2000). *Extreme programming explained: Embrace change.* Boston: Addison-Wesley.

Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., & Yi, W. (1995). UPPAAL: A tool suite for automatic verification of real-time systems. *Proceedings of Workshop on Verification and Control of Hybrid Systems III, 1066,* 232 – 243.

Bertolino, A. & Mirandola, R. (2004). Software performance engineering of component-based systems. *Proceedings of the 4th International Workshop on Software and Performance* (pp. 238 – 242). Redwood Shores, CA.

Binder, W. (September 2005). Portable, efficient, and accurate sampling profiling for java-based middleware. *Proceedings of the 5th international Workshop on Software Engineering and Middleware* (pp. 46 – 53), Lisbon, Portugal.

Box, D. & Shukla, D. (2006). WinFX workflow: Simplify development with the declarative model of windows workflow foundation. *MSDN Magazine, 21,* 54–62.

Bucci, G., Fedeli, A., Sassoli, L., & Vicario, E. (2003, July). Modeling flexible real time systems with preemptive time Petri nets. *Proceeding of 15th Euromicro Conference on Real-time Systems* (pp. 279 – 286). Porto, Portugal.

BPMN Information Home (2005). www.bpmn.org.

CPNTools: Computer tools for coloured Petri nets. (2006) Demark: University of Aarhus, CPN Group. wiki.daimi.au.dk/cpntools/cpntools.wiki.

Cantrill, B. M., Shapiro, M. W. & Levanthal, A. H. (2004). Dynamic instrumentation of production systems. *Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, Boston, MA.

Curtis, B., Kellner, M., & Over, J. (1992). Process modeling. *Communications of the ACM, 35* (9), 75 – 90.

Deng, G., Balasubramanian, J., Otte, W., Schmidt, D. & Gokhale, A. (2005). DAnCE: A QoS-enabled component deployment and configuration engine. *Proceedings of the 3rd Working Conference on Component Deployment*. Grenoble, France.

Florescu, O., Hoon, M., Voeten, J., & Corporall, H. (2006, July). Probabilistic Modelling and Evaluation of Soft Real-Time Embedded Systems. *Proceedings of Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI).* Samos, Greece.

Denning, P.J. & Buzen, J.P. (1978). The operation analysis of queuing network models. *Computing Surveys, 10,* 3, 225-261.

Fowler, M. (2006). *Continuous Integration*. www.martinfowler.com/articles/continuousIntegration.html

Garland, S. (2005). *The TIOA User Guide and Reference Manual.* tioa.csail.mit.edu/public/Documentation/ Guide.doc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, Addison-Wesley.

Gokhale, A., Balasubramanian, K., Balasubramanian, J., Krishna, A., Edwards, G., Deng, G., Turkay, E., Parsons, J., & Schmidt, D. (2006). Model driven middleware: A new paradigm for deploying and provisioning distributed real-time and embedded applications. *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*.

Grassi, V., Mirandola, R., & Sabetta, A. (2005). From design to analysis models: A kernel language for performance and reliability analysis of component-based systems. *Fifth International Workshop on Software and Performance*, Palma de Mallorca, Spain.

Hatcliff, J., Deng, W., Dwyer, M., Jung, G., & Prasad, V. (2003). Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*. Portland, OR.

Hauswirth, M., Diwan, A., Sweeney, P & Mozer, M. (2005). Automating vertical profiling. In *Proceeding of the 19th Conference of Object Oriented Programming, Systems, Languages and Applications*. San Diego, CA.

Herman, I., Melançon, G., & Marshall, M.S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics, 6* (1), 24 – 43.

Hill, J. H. & Gokhale, A. (2006). Continuous QoS provisioning of large-scale component-based systems using model driven engineering. Poster preseted at *International Conference on Model Driven Engineering Languages and Systems*, Genova, Italy.

Hill, J. H. & Gokhale, A. (2007, March). Model-driven engineering for development-time QoS validation of component-based software systems. In *Proceeding of International Conference on Engineering of Component Based Systems*, Tuscon, AZ.

IDC Quantitative Research Group (2005). *2005 Mission Critical Survey: Survey Report*. download.microsoft.com/download/1/8/a/18a10d4f-deec-4d5e-8b24-87c29c2ec9af/idc-ms-missioncritical-ww-261005.pdf

Karsai, G., Sztipanovits, J., Ledeczi, A. & Bapty, T. (2003). Model-integrated development of embedded software, In *Proceedings of the IEEE, 91* (1), 145-164.

Kaul, D., Kogekar, A., Gokhale, A., Gray, J., & Gokhale, S. (2007). Managing variability in middleware provisioning using visual modeling languages. In *Proceedings of the Hawaii International Conference on System Sciences HICSS-40 (2007), Visual Interactions in Software Artifacts Minitrack, Software Technology Track*. Big Island, HI.

Kristensen, L.M., Christensen, S., & Jensen, K. (1998). The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer, 2*, 98-132.

Lacour, S., Perez, C., & Priol, T. (2004). Deploying CORBA components on a computational grid: General principles and early experiments using the globus toolkit. In *Proceedings of the 2nd International Working Conference on Component Deployment*. Edinburgh, UK.

Lardieri, P., Balasubramanian, J., Schmidt, D., Thakar, G., Gokhale, A., & Damiano, T. (2007 to appear). A multi-layered resource management framework for dynamic resource management in enterprise DRE systems. In C. C Cavanaugh (Ed.) *Journal of Systems and Software: special issue on Dynamic Resource Management in Distributed Real-time Systems*.

Li, Z., Sun, W., Jiang, Z. B., & Zhang X. (2005) BPEL4WS unit testing: Framework and implementation. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)* (pp 103–110), Orlando, FL.

Ledeczi, A., Maroti, M., Karsai G., & Nordstrom G. (1999). Metaprogrammable toolkit for model-integrated computing. In *Proceedings of the IEEE International Conference on the Engineering of Computer-Based Systems Conference* (pp 311 – 317). Nashville, TN.

Lynch, N. and Tuttle, M. (1989). An introduction to input/output automata. *CWI-Quarterly, 2*(3), 219 – 246.

Margaria, T. (2004). Modeling dependable systems: What can model driven development contribute and what likely not? In *Proceedings of IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Vienna, Austria.

Mania, D., Murphy, J. & McManis, J. (2002). Developing performance models from non-intrusive monitoring traces. In *Proceeding of Information Technology and Telecommunications (IT&T).*

Merlin , P. M (1974). A study of the recoverability of computing systems. Department of Information and Computer Science, University of California, Irvine, CA.

Menascé, D., Almeida, V. & Dowdy, L. (2004). *Performance By Design: Computer Capacity Planning by Example* (pp. 135 – 141). Prentice Hall: Upper Saddle River, NJ.

Dutto, M. & Lautenbacher, F. (2007). *Java Workflow Tooling (JWT) Creation Review.* www.eclipse.org/proposals/jwt/JWT%20Creation%20Review%2020070117.pdf

Object Management Group (2002). *Real-time CORBA Specification.* OMG Document formal/02-08-02.

Northrop, L., Feiler. P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K. & Wallnau, K. (2006). *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon.

Parsons, T., Mos, A. & Murphy, J. (2006, August). Non-intrusive end-to-end runtime path tracing for J2EE systems. *IEEE Proceedings Software, 153*, 149 – 161.

Peterson, James L. (1977). Petri nets. *ACM Computing Surveys, 9* (3), 223 – 252.

PlanetLab Consortium (2006). *PLANETLAB: An Open Platform for Developing, Deploying, and Accessing Planetary-Scale Services*. http://www.planet-lab.org.

Ricci, R., Alfred, C., & Lepreau, J. (2003). A solver for the network testbed mapping problem. *SIGCOMM Computer Communications Review, 33*.

Shankaran, N., Koutsoukos, X., Schmidt, D.C., & Gokhale, A. (2005). Evaluating adaptive resource management for distributed real-time embedded systems. In *Proceedings of 4th Workshop on Adaptive and Reflective Middleware*. Grenoble, France.

Slaby, J., Baker, S., Hill, J. & Schmidt, D. (2005). Defining behavior and evaluating QoS performance of the SLICE scenario (Tech. Rep. No. ISIS-05-608). Nashville, TN: Vanderbilt University.

Slaby, J., Baker, S., Hill, J. & Schmidt, D. (2006). Applying system execution modeling tools to evaluate enterprise distributed real-time and embedded system QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*. Sydney, Australia.

Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer, 39*, 41 – 47.

Smith, C. (1990). *Performance Engineering of Software Systems*. Addison-Wesley.

Smith, C. & Williams, L. (2001). *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley.

Smith, M., Friese, T., & Freisleben, B. (2006). Model driven development of service-oriented grid applications. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*. Guadeloupe, French Caribbean.

Subramonian, V. (2006). *Timed automata models for principled composition of middleware* (Tech. Rep. No. WUCSE-2006-23). St. Louis, MI: Washington University, Computer Science and Engineering Department.

Task, B., Paniscotti, D., Roman, A., & Bhanot, V. (2006). Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications. In *Proceedings of International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. Portland, OR

Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, K., Chase, J., & Becker, D. (2002). Scalability and accuracy in a large-scale network emulator. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*.

Waddington, D., Roy, N. & Schmidt, D.C. (2007). Dynamic analysis and profiling of multi-threaded systems. *Designing Software-Intensive Systems: Methods and Principles*, Ed. Dr. Pierre F. Tiako, Langston University, OK.

Wang, N. & Gill, C. (2003). Improving real-time system configuration via a QoS-aware CORBA component model. *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems*.

White, J., Schmidt, D.C., & Gokhale, A. (2005). Simplifying the development of autonomic enterprise Java Bean applications via model driven development. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*. Seattle, WA.

White, J., Gokhale, A. & Schmidt, D.C. (2007). Simplifying autonomic enterprise Java Bean applications via model-driven development: A case study. Submitted to the *Journal of Software and System Modeling*.

White, S. A. (2005, March). Using BPMN to Model a BPEL Process. www.bptrends.com.

Yilmaz, C., Krishna, A. S., Memon, A., Porter, A., Schmidt, D. C., Gokhale, A. & Natarajan, B. (2005). Main effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. *Proceedings of the 27th International Conference on Software Engineering, 293 – 302*, St. Louis, MO.

ADDITIONAL READING

Chatterjee, A. (2007). Service-component architectures: A programming model for SOA. *Dr. Dobb's Journal, 400*, 40 – 45.

Chilimbi, T. M. & Hauswirth, M. (2004). Low-overhead memory leak detection using adaptive statistical profiling. *Proceedings of the 11th international Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA.

Cascaval, C., Duesterwald, E., Sweeney, P. F., & Wisniewski, R. W. (2006). Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development, 50* (2/3), 239 – 248.

Haran, M., Karr, A., Orso, A., Portor, A. & Sanil, A. (2005). Applying classification techniques to remotely-collected program execution data. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal.

Hauswirth, M., Sweeney, P., Diwan, A. & Hind, M. (2004). Vertical profiling: Understanding the behavior of object-oriented applications. *ACM SIGPLAN Notices, 39* (10), 251 – 269.

Hill, J. H. & Gokhale, A. (in press). Model-driven engineering for early QoS validation of component-based software systems, *Journal of Software, 2* (2).

Huselius, J. & Andersson, J. (2005). Model synthesis for real-time systems. *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, Manchester, UK.

Laugelier, G., Sahraoui, H., & Poulin, P. (2005). Visualization-based analysis of quality for large-scale software systems. *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, Long Beach, CA.

Li, Z., Sun, W., Jiang, Z. B., & Zhang, X. (2005). BPEL4WS unit testing: Framework and implementation. *Proceedings of the IEEE International Conference on Web Services*, Orlando, FL.

Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., Maroti, M. (2001). On metamodel composition. *Proceedings of the 2001 IEEE International Conference on Control Applications*, Mexico City, Mexico.

Kaynar, D. K., Lynch, N., Segala, R., & Vaandrager, F. (2006). *The Theory of Timed I/O Automata, Synthesis Lectures on Computer Science.* Morgan and Claypool Publishers.

Kounev, S. & Buchmann, A. (2003). Performance modeling and evulation of large-scale J2EE applications. *Proceedings of the 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems*, Dallas, TX.

Kounev, S. (2006). Performance modeling and evaluation of distributed component-based systems using queuing Petri nets. *IEEE Transactions of Software Engineering, 32* (7), 486 – 502.

Memon, A., Porter, A., Nagarajan, A., Schmidt, D. & Natarajan, B. (2004). Skoll: Distributed quality assurance. *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland.

Metz, E., Lencevicius, R., & Gonzalez, T. (2005). Performance data collection using a hybrid approach. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal.

Mos A. & Murphy, J. (2004). COMPAS: Adaptive Performance Monitoring of Component-Based Systems. *Proceedings of 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems*, Beijing, China.

Odom, J., Hollingsworth, J. K., DeRose, L., Ekanadham, K., & Sbaraglia, S. (2005). Using dynamic tracing sampling to measure long running programs. *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Seattle, WA.

Parsons, T. & Murphy, J. (in press). Detecting performance antipatterns in component-based enterprise systems. *Journal of Object Technology*.

Saff, D. & Ernst, M. D. (2004). An experimental evaluation of continuous testing during development. *Proceedings of the 2004 ACM SIGSOFT international Symposium on Software Testing and Analysis*, Boston, MA.

Schroeder, P. J., Kim, E., Arshem, J., Bolaki, P. (2003). Combining behavior and data modeling in automated test case generation. *Proceedings of the 3rd International Conference on Quality Software*, Dallas, TX.

Srinivas, K. & Srinivasan, H. (2005). Summarizing application performance from a components perspective. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal.

Stewart, C. & Shen, K. (2005). Performance modeling and system management for multi-component online services. *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA.

Wu, W., Spezialetti, M. & Gupta, R. (1996). Designing a non-intrusive monitoring tool for developing complex distributed applications. *Proceedings of the 2nd IEEE international Conference on Engineering of Complex Computer Systems*, Washington, D.C.

BIOGRAPHIES

James H. Hill is a 4th year Ph.D. student in the Electrical Engineering and Computer Science Department at Vanderbilt University, Nashville, TN. His primary research interests include using model-driven engineering techniques to assist in locating flaws related to quality-of-service earlier in the development lifecycle as opposed to integration time when it can require more time and effort to locate and resolve them. He received his B.S. in Computer Science from Morehouse College, Atlanta, GA in 2004 and M.S. in Computer Science from Vanderbilt University in 2006. James Hill is a member of ACM.

Douglas C. Schmidt is a Professor of Computer Science and Associate Chair of the Computer Science and Engineering program at Vanderbilt University. He has published 9 books and over 350 technical papers that cover a range of research topics, including patterns, optimization techniques, and empirical analyses of software frameworks and domain-specific modeling environments that facilitate the development of distributed real-time and embedded (DRE) middleware and applications running over high-speed networks and embedded system interconnects. Dr. Schmidt has also led the development of ACE, TAO, CIAO, and CoSMIC, which are widely used, open-source DRE middleware frameworks and model-driven tools that contain a rich set of components and domain-specific languages that implement patterns and product-line architectures for high-performance DRE systems.

John M. Slaby is currently a Senior Principle S/W Engineer with Raytheon Integrated Defense Systems. He has spent over 25 years in software engineering working for high technology start-ups focused on data networking. He is currently involved in research focused on dynamic resource management and model-based engineering using domain-specific modeling languages. His background includes software engineering, engineering management, architecture, product management, product marketing, training, and customer support.